UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

JOSÉ CARLOS SANT'ANNA PALMA

Reduzindo o Consumo de Potência em Redes Intra-Chip através de Esquemas de Codificação de Dados

Tese apresentada como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação

Prof. Dr. Ricardo Augusto da Luz Reis
Orientador

Prof. Dr. Fernando Gehm Moraes
Co-orientador

Porto Alegre, setembro de 2007.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Palma, José Carlos Sant'Anna Palma

Reduzindo o Consumo de Potência em Redes Intra-Chip através de Esquemas de Codificação de Dados / José Carlos Sant'Anna Palma – Porto Alegre: Programa de Pós-Graduação em Computação, 2007.
152 f.:il.

Tese (doutorado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2007. Orientador: Ricardo Augusto da Luz Reis; Co-orientador: Fernando Gehm Moraes.

1. Systemas-em-Chip. 2. Arquiteturas de comunicação. 3. Redes Intra-Chip. 4. Esquemas de Codificação de Dados. I. Reis, Ricardo Augusto da Luz. II. Moraes, Fernando Gehm. III. Título.

AGRADECIMENTOS

Ao CNPq e à CAPES, por terem viabilizado este trabalho através do suporte financeiro.

Ao PPGC, pelas instalações oferecidas para o desenvolvimento do trabalho.

Aos funcionários do PPGC, pela disposição para resolver assuntos burocráticos.

Ao corpo docente do PPGC, pela amizade formada no decorrer do curso. Dentre estes, gostaria de citar os professores Altamiro Susin, Sérgio Bampi, Marcelo Johann e Flávio Wagner.

Aos colegas do grupo GME, os quais foram de grande importância na troca de conhecimento e no convívio diário.

Ao meu orientador, Ricardo Augusto da Luz Reis, e ao meu co-orientador, Fernando Gehm Moraes, pelo companheirismo, amizade, e apoio durante o desenvolvimento do trabalho.

Ao meu ex-orientador de bolsa IC e de trabalho de conclusão de curso durante a graduação, Leandro Soares indrusiak, e à sua família, pela amizade e apoio durante os meses de estágio na Alemanha, tanto no desenvolvimento do trabalho quanto no convívio diário.

À minha família, que mesmo estando distante, me incentivou e acreditou no meu sucesso, muitas vezes mais do que eu.

Ao meu avô, que sempre me apoiou, sempre demonstrou ter orgulho do neto, mas que infelizmente faleceu poucos meses antes do término deste trabalho.

À minha namorada que, mesmo com pouco tempo de convívio, foi de grande importância na reta final do desenvolvimento deste trabalho.

A todos que, direta ou indiretamente, contribuíram para o desenvolvimento deste trabalho e para o meu crescimento pessoal.

A Deus, pela minha família, pelos meus amigos, pela oportunidade de aumentar meus conhecimentos e por conhecer pessoas especiais.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

AHB	*Advanced High-performance Bus*
AMBA	*Advanced Microcontroller Bus Architecture*
APB	*Advanced Peripheral Bus*
APB_F	*Average Power per Buffer*
APD	*Average Power per Decoder*
APE	*Average Power per Encoder*
APH	*Average Power per Hop*
APL_R	*Average Power per Router Link*
APL_L	*Average Power per Local Link*
APR	*Average Power per Router*
APS_C	*Average Power per Switch Control*
CI	Circuito Integrado
CAD	*Computer Aided Design*
CPU	*Central Processing Unit*
DCR	*Device Control Register*
DMA	*Direct Memory Access*
GALS	*Globally Asynchronous Locally Synchronous*
ISO	*International Organization for Standardization*
ITRS	*International Technology Roadmap for Semiconductors*
IP	*Intellectual Property (core)*
NoC	*Network-on-Chip*
OPB	*On-Chip Peripheral Bus*
OSI	*Open System Interconnection*
PCI	*Peripheral Component Interconnect*
PLB	*Processor Local Bus*
SoC	*System-on-Chip*
SPIN	*Scalable Programmable Integrated Network*

UART *Universal Asynchronous Receiver Transmitter*

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

O consumo de potência em uma Rede Intra-Chip (em inglês, *Network-on-Chip* – NoC) cresce linearmente com a quantidade de transições de sinais nos pacotes transmitidos através da infra-estrutura de interconexão. Uma forma de minimizar o consumo de potência em um sistema baseado em NoC é reduzir a atividade de transição de sinais nas portas de entrada dos módulos que constituem a NoC. Esta redução pode ser obtida através da utilização de esquemas de codificação de dados. Vários esquemas de codificação foram propostos no final dos anos 90, porém direcionados a arquiteturas de comunicação baseadas em barramentos. Este trabalho investiga a utilização destes esquemas de codificação em sistemas baseados em *Networks-on-Chip*. Dentre os esquemas encontrados na literatura, quatro foram implementados e avaliados neste trabalho. Este trabalho também apresenta como contribuição original um novo esquema de codificação de dados adequado a NoCs.

A estimativa do consumo de potência da NoC é calculada com base em macromodelos que reproduzem a potência consumida em cada módulo interno da NoC, de acordo com a atividade de transição de sinais no tráfego recebido. Estes macromodelos são aqui caracterizados através da simulação elétrica de cada módulo da NoC e dos esquemas de codificação. Para permitir a análise de consumo com tráfegos de aplicações reais, os macromodelos são inseridos em um modelo de mais alto nível de abstração. Este modelo é empregado para analisar o balanço entre redução de potência obtida com a redução da transição de sinais e o consumo extra do esquema de codificação.

A maioria dos esquemas de codificação encontrados na literatura reduz efetivamente a atividade de transição de sinais. Porém, o impacto do consumo extra de potência para codificar e decodificar os dados não é avaliado. A avaliação conduzida neste trabalho considera o consumo da codificação/decodificação em uma NoC real, quantificando a redução de consumo obtido com cada esquema de codificação. Devido ao baixo desempenho dos esquemas de codificação existentes, quando aplicados a NoCs, foi desenvolvido um novo esquema, chamado *T-Bus-Invert*. Os resultados mostram um desempenho superior do *T-Bus-Invert* quando comparado aos demais esquemas para *flits* com largura de 8 e 16 bits, e um desempenho similar ao do *Bus-Invert* com 4 *clusters* para *flits* de 32 bits.

Palavras-Chave: Systemas-em-Chip, Arquiteturas de comunicação, Redes Intra-Chip, Esquemas de Codificação de Dados.

Reducing the Power Consumption in Networks-on-Chip through Data Coding Schemes

ABSTRACT

The power consumption in Networks-on-Chip grows linearly with the amount of signal transitions in successive data packets sent through this interconnection infrastructure. One option to decrease the power consumption in NoC-based systems is reducing the switching activity at the input ports of NoC modules. This reduction can be achieved by means of data coding schemes. Several schemes were proposed in the nineties. However, all of them address only bus-based communication architectures. This work investigates the use of such data coding schemes in NoC-based systems. Among the coding schemes found in the literature, four were implemented and evaluated in this work. This work also presents a new data coding scheme, named *T-Bus-Invert*, suitable for NoCs.

Estimations of the NoC power consumption are computed here based on macromodels which reproduce the power consumption on each internal NoC module, according to the transition activity in the input traffic. Such macromodels are characterized through electrical simulations of each NoC module and coding circuits. To enable the evaluation of real applications traffic, such macromodels are inserted in a higher abstraction level model. This model is employed to analyze the trade-off between the power saving due to coding schemes versus the power consumption overhead due to the encoding and decoding modules.

Most of the coding schemes proposed in the literature effectively reduce the switching activity, but the overall impact of the power consumption to encode/decode data in the system is not evaluated. The evaluation conducted in this work considers the power consumption to encode/decode data in a real NoC, quantifying the power savings for each coding scheme. Due to the insufficient performances of the existing schemes when applied to NoCs, a coding scheme, *T-Bus-Invert*, was developed. Results showed superior performance of the *T-Bus-Invert* compared to all evaluated coding schemes for 8 and 16-bit *flits*, and similar performance to the 4-cluster *Bus-Invert* for 32-bit *flits*.

Keywords: Systems-on-Chip, Communication Architectures, Networks-on-Chip, Data Coding Schemes.

1 INTRODUÇÃO

O desenvolvimento de novas tecnologias na fabricação de circuitos integrados (CIs) permite a implementação de sistemas complexos, com milhões de transistores, em uma única pastilha de silício, chamados de *Systems-on-Chip* (SoCs). O ritmo desses avanços da tecnologia de fabricação tem se mantido exponencial nas últimas décadas, segundo a Lei de Moore (SCHALLER, 1997). Esta lei é devida a Gordon E. Moore, que em 1965 observou que a densidade de componentes em circuitos integrados dobrava a intervalos regulares, inferindo que este comportamento perduraria por muito tempo ainda. O intervalo medido por Moore para que a densidade média dos circuitos integrados (CIs) dobrasse foi de 18 meses (CALAZANS, 1998). Em 1970 ele estendeu o intervalo para 24 meses, o que ainda hoje permanece uma taxa estável, podendo perdurar por mais 10 a 15 anos (ITRS, 2005).

A vantagem na utilização de SoCs está na interface de comunicação, pois nos sistemas atuais o maior gargalo é a perda de desempenho causada pela troca de informações entre o hardware e o software executados em CIs distintos. Caso os componentes de hardware e software estejam integrados em um único CI, o desempenho global do sistema tende a ser muito maior. Além do mais, em um mercado caracterizado não apenas pela elevada complexidade dos sistemas e seu alto desempenho, mas também por um curto *time-to-market* (tempo decorrido entre a especificação de um novo produto e a chegada do produto ao mercado consumidor) e baixo consumo de potência, a possibilidade de produzir-se um SoC torna possível atender às pressões do mercado e amortizar os custos de projeto entre vários sistemas. Para isto, é necessário que os componentes integrados em um SoC sejam reutilizáveis, evitando que os mesmos tenham que ser novamente projetados. Dessa forma, as metodologias de projeto adotadas devem ser baseadas no reuso de núcleos de hardware (DESIGN-REUSE, 2003), denominados núcleos de propriedade intelectual, em inglês, IP *Cores* (*Intellectual Property Cores*) ou simplesmente IPs (GUPTA; ZORIAN, 1997; RINCON et al., 1997).

Um núcleo de hardware é um módulo complexo, digital ou analógico, podendo ser descrito em diferentes níveis de abstração (PALMA, 2002). Estes núcleos são pré-projetados, pré-verificados e prototipados em hardware pelo menos uma vez (PALMA, 2000). Podem ser desenvolvidos pela empresa responsável pelo projeto do sistema ou adquiridos de terceiros. Desta forma, o projetista pode concentrar-se no sistema sem ter que se preocupar com a funcionalidade interna ou com o desempenho de componentes individuais. Conforme as estimativas da indústria de semicondutores, o percentual de reuso em CIs será de 90% em 2012 (SIA, 2005).

Em um SoC, os núcleos são interconectados por uma arquitetura de comunicação (MADISETTI; SHEN, 1997). Duas abordagens são utilizadas na implementação desta arquitetura de comunicação nos SoCs convencionais: canais ponto-a-ponto dedicados e

canais multiponto compartilhados, como mostra a Figura 1.1. Canais ponto-a-ponto oferecem o melhor desempenho, pois cada comunicação ocorre independentemente das demais por meio de canais exclusivos. Porém, ela requer um projeto específico e, portanto, possui reusabilidade limitada. Já na arquitetura multiponto, mais conhecida como barramento, a mesma estrutura pode ser reutilizada em diferentes sistemas, reduzindo o tempo de projeto (GUERRIER; GREINER, 2000-b). Tipicamente, a arquitetura de comunicação utilizada é o barramento (ou uma hierarquia com dois ou mais barramentos), pois oferece como vantagem características de reusabilidade e baixo custo de silício (ZEFERINO, 2003).

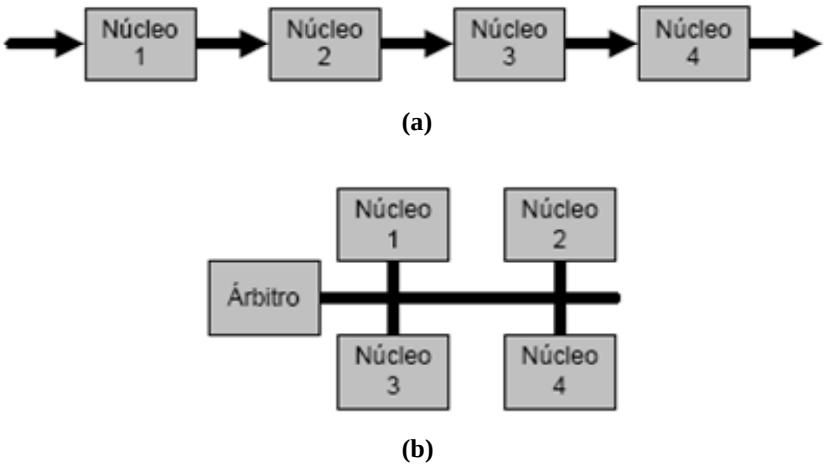


Figura 1.1: Redes de interconexão nos SoCs atuais: (a) ponto-a-ponto; (b) multi-ponto (ZEFERINO, 2003).

Existem algumas arquiteturas de barramento disponíveis no mercado de semicondutores para guiar os fabricantes, tais como a *CoreConnect* (IBM, 2000) da *IBM, AMBA* (IBM, 2003) da *ARM* e *Wishbone* (SILICORE, 2003; OPENCORES.ORG, 2003) da *Silicore*. Estas arquiteturas de barramento são geralmente vinculadas à arquitetura de um processador, tal como o *PowerPC* ou o *ARM* (BERGAMASCHI; LEE, 2000).

A interconexão por barramento é simples, sob o ponto de vista de implementação, apresentando, entretanto, diversas desvantagens (BENINI; DE MICHELI, 2001): (*i*) apenas uma troca de dados pode ser realizada por vez em cada barramento, pois o meio físico é compartilhado por todos os núcleos de hardware, reduzindo o desempenho global do sistema (LIANG; SWAMINATHAN; TESSIER, 2000); (*ii*) necessidade de mecanismos inteligentes de arbitragem do meio físico para evitar desperdício de largura de banda (HU; DENG; MARCULESCU, 2002); (*iii*) a escalabilidade é limitada, ou seja, o número de núcleos de hardware que podem ser conectados a cada barramento é baixo, tipicamente na ordem de dezenas (HWANG, 1993); (*iv*) o uso de linhas globais em um circuito integrado com tecnologia submicrônica impõe sérias restrições ao desempenho do sistema devido às altas capacitâncias e resistências parasitas inerentes aos fios longos (RABAEY, 1996; LANGEN; BRINKMANN; RUCKERT, 2000).

Além disso, os efeitos físicos da tecnologia submicrônica tornam cada vez mais difícil manter a sincronia global entre todas as partes do circuito integrado. O sinal de *clock* logo precisará de vários ciclos para atravessar o circuito e o escorregamento (*skew*) do *clock* tornar-se-á intratável, devido ao crescimento significativo da árvore de distribuição do mesmo, que já é hoje a maior fonte de consumo de energia (JANTSCH; TENHUNEM, 2003).

Quanto à potência, o problema do barramento é que cada sinal deve chegar a todos os pontos do mesmo, exigindo uma grande quantidade de energia. Vários autores (KUMAR et al., 2002; BENINI; DE MICHELI, 2002; GUERRIER; GREINER, 2000-b; DALLY; TOWLES, 2001; RIJPKEMA, 2001; SGROI et al., 2001) concordam que as interconexões físicas no circuito integrado serão o fator limite para o desempenho e, possivelmente, para o consumo de energia nos futuros SoCs.

Devido a estas desvantagens, muitos projetistas têm proposto uma mudança partindo do paradigma de projeto totalmente sincronizado, para um novo paradigma de projeto globalmente assíncrono e localmente síncrono (GALS – *Globally Asynchronous, Locally Synchronous*) (BENINI; DE MICHELI, 2002). O paradigma GALS subdivide a aplicação em sub-aplicações. Cada sub-aplicação corresponde a um projeto físico síncrono localizado em um *tile1*, enquanto que a comunicação entre os *tiles* é realizada através de um recurso de comunicação assíncrono.

Uma rede intra-chip (em inglês, *Network-on-Chip* – NoC) é uma infra-estrutura essencialmente composta por roteadores interconectados por canais de comunicação. Uma NoC pode oferecer uma comunicação assíncrona, o que a torna especialmente adequada para lidar com o paradigma GALS. Outras vantagens oferecidas pelas NoCs são: paralelismo, alta escalabilidade, reusabilidade e confiabilidade (DALLY; TOWLES, 2001; WINGARD, 2001). Como exemplos de NoCs pode-se citar a arquitetura de conexão SPIN (*Scalable Programmable Integrated Network*) (GUERRIER; GREINER, 1999) (com resultados experimentais apresentados em (GUERRIER, 2000-a) e (GUERRIER, 2000-b), a arquitetura aSOC (LIANG; SWAMINATHAN; TESSIER, 2000), a rede CLICHÉ KUMAR et al., 2002), a rede SoCIN (ZEFERINO, 2003) e a rede Hermes (MORAES et al., 2004).

1.1 Motivação

O crescimento do mercado para dispositivos portáteis alimentados por bateria reforça a importância da redução de potência na exploração do espaço de projeto, antes voltado principalmente para área, desempenho e testabilidade (SINGH et al., 1995; BURD; BRODERSEN, 2002). O consumo de potência influencia diretamente na duração da carga da bateria, bem como nos requisitos de encapsulamento e dissipação de calor (PEDRAM, 1996). A fim de garantir que o sistema final esteja de acordo com os requisitos funcionais, térmicos e de custo desejados, a questão do consumo de potência deve ser levada em consideração durante o projeto de todos os subsistemas em um SoC, incluindo a estrutura de interconexão.

---

1 Os núcleos de uma NoC são posicionados dentro de regiões isócronas, chamadas de *tiles*.

Um problema relacionado ao consumo de potência nos barramentos são as capacitâncias induzidas pelas linhas de comunicação longas. Este problema é minimizado em NoCs, já que esta abordagem utiliza linhas de comunicação ponto-a-ponto não-globais entre roteadores. Entretanto, NoCs consomem potência nos roteadores, reduzindo a vantagem aparente em termos de consumo de potência em comparação com barramentos.

Conforme será mostrado na seção 4.3 deste trabalho, o consumo de potência em uma NoC cresce linearmente com a quantidade de transições de sinais ocasionadas pelos pacotes transmitidos através da arquitetura de comunicação (PALMA et al., 2005). Uma forma de reduzir o consumo de potência em um sistema baseado em NoC é posicionando próximos os núcleos que trocam muitas mensagens entre si. Além disso, também é importante que se leve em consideração a taxa de transição de sinais destas mensagens, visto que a omissão desta informação pode levar a um erro de mais de 100% na avaliação do consumo de potência na NoC (MARCON et al., 2005-c). Utilizando a rede Hermes (MORAES et al., 2004) como estudo de caso, este trabalho mostra que as transições de bit podem afetar o consumo de potência em mais de 370% nas linhas de interconexão, 180% nos buffers de entrada dos roteadores e 16% na lógica de controle dos roteadores.

Sendo assim, outra forma de reduzir o consumo de potência na NoC, tanto na lógica quanto nas interconexões, é reduzindo a atividade de transição de sinais nas portas de entrada dos módulos da mesma. Esta redução pode ser feita através da utilização de esquemas de codificação de dados. Vários esquemas de codificação foram propostos no final dos anos 90, direcionados a arquiteturas de comunicação baseadas em barramentos.

1.2 Objetivos e Metodologia

Este trabalho investiga a utilização destes esquemas de codificação de dados no contexto de sistemas baseados em NoCs, analisando o compromisso entre redução de potência obtida com a redução da transição de sinais e o consumo extra do esquema de codificação.

A estimativa do consumo de potência é feita com a utilização de macromodelos que reproduzem a potência consumida em cada módulo interno da NoC, de acordo com a atividade de transição de sinais no tráfego recebido. Estes macromodelos são construídos com base na simulação SPICE dos módulos da NoC e dos módulos que implementam os esquemas de codificação.

Para chegar até a simulação SPICE, é proposto um fluxo de projeto partindo da descrição VHDL dos módulos, passando pela simulação e síntese lógicas, até a descrição SPICE, que é simulada eletricamente. Na simulação lógica são produzidos os estímulos de entrada para os módulos que, após a síntese lógica, são convertidos para um netlist SPICE, descrevendo portas lógicas e suas interconexões. Estas portas lógicas, por sua vez, são descritas em transistores em uma biblioteca que também serve de entrada para o simulador SPICE.

Os macromodelos de potência são embarcados em um modelo de mais alto nível e uma série de simulações são executadas, com o objetivo de analisar o balanço entre redução de potência obtida com a redução da transição de sinais e o consumo extra do esquema de codificação.

Este trabalho apresenta também um novo esquema de codificação de dados eficiente para NoCs, o *T-Bus-Invert*. Conforme os experimentos desenvolvidos no escopo deste trabalho, este esquema de codificação foi mais eficiente do que outros quatro encontrados na literatura e implementados, utilizando-se largura de *flit* igual a 8 e 16 bits. Na NoC com largura de *flit* igual a 32 bits o esquema *T-Bus-Invert* foi o segundo melhor, mas mantendo a mesma eficiência apresentada nos outros estudos-de-caso.

1.3 Contribuições do Trabalho

Pode-se apontar como principais contribuições deste trabalho:

- A análise do efeito da transição de sinais sobre o consumo de potência na NoC;
- A implementação de diferentes esquemas de codificação encontrados na literatura, inserindo-os em um sistema baseado em NoC;
- As alterações da NoC, a fim de adaptá-la ao esquema de codificação, quando necessário;
- O fluxo para aquisição dos parâmetros de potência;
- O desenvolvimento de ferramentas de conversão de níveis de descrição utilizadas no fluxo;
- A criação de macromodelos de consumo de potência para os módulos da NoC com diferentes configurações e para os módulos dos esquemas de codificação com diferentes larguras de *flit*.
- O método utilizado para estimar o consumo de potência de acordo com a percentagem de transições de sinais no tráfego recebido;
- A análise de diferentes esquemas de codificação, quando empregados em diferentes configurações da NoC, utilizando padrões de tráfego reais;

  A implementação de um novo esquema de codificação.

1.4 Organização do Trabalho

Este trabalho está organizado como segue. O capítulo 2 apresenta uma revisão sobre os conceitos básicos de *Networks-on-Chip*. No final deste capítulo é realizada a descrição das características principais da rede Hermes, utilizada como estudo de caso deste trabalho.

No capítulo 3 são apresentados diferentes esquemas de codificação de dados encontrados na literatura. Alguns destes esquemas foram implementados no sistema baseado em NoC, enquanto outros são inviáveis, pois exigem um aumento significativo nos canais de comunicação e nos módulos da NoC. Este capítulo apresenta também um novo esquema de codificação, o *T-Bus-Invert*, uma das contribuições deste trabalho. Ao final deste capítulo é introduzida uma abordagem de sistema baseado em NoC integrando ao mesmo esquemas de codificação de dados.

O capítulo 4 apresenta o modelo de consumo de potência em NoCs, proposto no escopo deste trabalho, explicando como foi feita a definição deste modelo, bem como a obtenção dos parâmetros de potência. Também neste capítulo são apresentados os macromodelos de potência para cada módulo da NoC e dos esquemas de codificação.

Estes macromodelos são utilizados para estimar o consumo de cada módulo de acordo com a atividade de transição em seus sinais de entrada.

No capítulo 5 é feita a análise do consumo de potência da NoC com e sem codificação, apresentando resultados experimentais obtidos através de simulações com diferentes tipos de tráfego. A análise é feita com base nos macromodelos de potência apresentados no capítulo 4.

O capítulo 6 apresenta as conclusões deste trabalho.

No anexo são apresentados os códigos VHDL de alguns dos módulos de codificação e decodificação implementados neste trabalho. Os códigos não apresentados são semelhantes aos que se encontram em anexo, variando apenas a largura de *flit*.

2 NETWORKS-ON-CHIP

A idéia em uma *Network-on-Chip* é separar a estrutura de comunicação e a aplicação também no *layout* físico. A única limitação aos recursos (unidade computacional ou de armazenamento) está relacionada ao seu tamanho e sua interface. Qualquer tipo de recurso que possa ser implementado em uma área síncrona (que utiliza um mesmo *clock*) pode ser conectado à NoC (SOININEN; HEUSALA, 2003). Do ponto de vista do sistema, cada recurso é um sistema embarcado independente que possui uma interface padronizada de acordo com a estrutura de comunicação. Um sistema baseado em NoC pode ser visto como um sistema distribuído, com recursos que podem utilizar diferentes domínios de *clock* e se comunicar entre si síncrona ou assincronamente (paradigma GALS – *Globally Asynchronous Locally Synchronous*).

Espera-se que os sistemas baseados em NoCs forneçam boas soluções para o reuso de núcleos de hardware (KUMAR et al., 2002), já que as NoCs possuem as seguintes características: (*i*) eficiência no consumo de energia (BENINI; DE MICHELI, 2001); (*ii*) largura de banda escalável, quando comparada à arquiteturas de barramento tradicionais; (*iii*) reusabilidade; (*iv*) decisões de roteamento distribuídas (GUERRIER; GREINER, 2000-b); (*v*) paralelismo na comunicação.

Embora tenham como desvantagem um custo maior em área e latência na comunicação, esses problemas podem ser atenuados pela grande disponibilidade de transistores e por soluções arquiteturais que permitem reduzir a latência da rede e seus efeitos no desempenho da aplicação (ZEFERINO, 2003).

As *Networks-on-Chip* baseiam-se nas redes de interconexão chaveadas utilizadas em computadores paralelos e, portanto, herdam os conceitos destas redes. A seção 2.1 apresenta os conceitos básicos de *Networks-on-Chip*, descrevendo os nodos de chaveamento e de processamento, bem como enlaces, mensagens e pacotes e, por fim, as características de uma NoC. Estas características são detalhadas nas seções 2.2 a 2.7. A seção 2.8 descreve três casos que impedem que a comunicação seja realizada (*starvation*, *livelock* e *deadlock*) e, portanto, devem ser evitados.

Como exemplos de *Networks-on-Chip* pode-se citar a rede SoCIN (ZEFERINO, 2003), desenvolvida pelo Grupo de Microeletrônica (GME) da UFRGS, e a rede Hermes (MORAES et al., 2004), desenvolvida pelo Grupo de Apoio ao Projeto de Hardware (GAPH) da PUCRS. A rede Hermes foi escolhida como estudo de caso neste trabalho devido à possibilidade de geração automatizada de código VHDL para sua descrição, com diferentes configurações, bem como geração automática de *testbenches* (MELLO et al., 2005). A seção 2.9 apresenta as características da rede Hermes,

juntamente com o ambiente ATLAS (MELLO et al., 2005), ferramenta que automatiza os vários processos relacionados ao fluxo de projeto da rede Hermes.

2.1 Conceitos Básicos de NoCs

Uma NoC consiste de uma rede composta por nodos de processamento (recursos) conectados a um nodo de chaveamento, e de nodos de chaveamento conectados a outros nodos de chaveamento através de canais de comunicação. A Figura 2.1 (MELLO; MÖLLER, 2003) ilustra uma NoC com topologia (arranjo dos nodos e canais sob a forma de um grafo) em anel. Esta é uma topologia bastante simples e econômica quanto à forma como são interconectados os nodos de chaveamento. Cada nodo de chaveamento possui ligações para dois nodos de chaveamento vizinhos e para um nodo de processamento local.

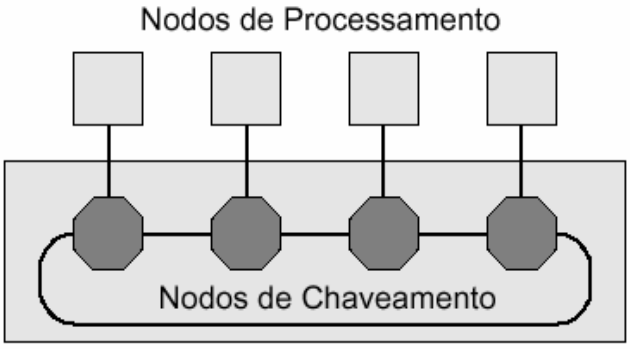


Figura 2.1: Exemplo de NoC com topologia em anel (MELLO; MÖLLER, 2003).

2.1.1 Nodos de Processamento e Nodos de Roteamento

Os nodos de processamento (Figura 2.2a) são responsáveis pela execução das tarefas do sistema, enquanto que os nodos de chaveamento (Figura 2.2b, também chamados de roteadores) são responsáveis pela transferência de mensagens entre nodos de processamento. Em geral, os nodos de chaveamento possuem um núcleo de chaveamento (ou chave), uma lógica para roteamento e arbitragem (abreviado por R&A na Figura 2.2b) e portas de comunicação para outros nodos de chaveamento e, dependendo da topologia, para um ou mais nodos de processamento locais. As portas de comunicação incluem canais de entrada e de saída, os quais podem possuir, ou não, *buffers* para o armazenamento temporário de informações. Os nodos de processamento devem possuir uma interface de comunicação compatível com a NoC (interface de rede). Quando se reutiliza núcleos de hardware prontos, muitas vezes é necessária a utilização de *wrappers* com adaptadores de interface, para tornar a interface do núcleo compatível com a da NoC.

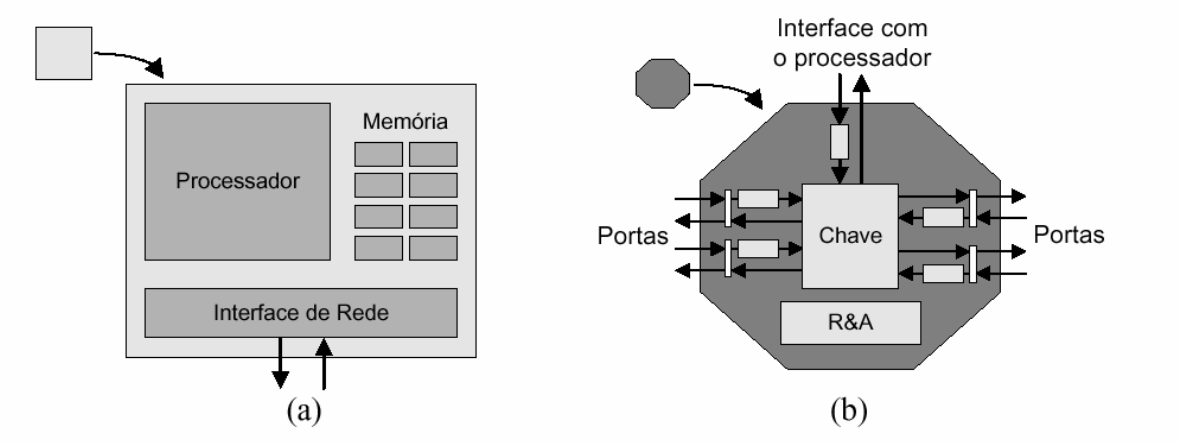


Figura 2.2: Exemplos de nodos (a) de processamento e (b) de chaveamento (MELLO; MÖLLER, 2003).

2.1.2 Enlaces

Chama-se enlace (ou *link*) a ligação física entre dois nodos de chaveamento. Dependendo da topologia da rede, um nodo de processamento e um nodo de chaveamento também podem ser interligados através de um enlace. Cada canal físico pode conter, além dos sinais de dados (*data*), sinais de controle de fluxo (*tx* e *ack*), sinais de enquadramento de mensagem (*bp* e *ep*), bem como sinais de paridade (*par*) e sinalização de erro (*er*), como mostra a Figura 2.3.

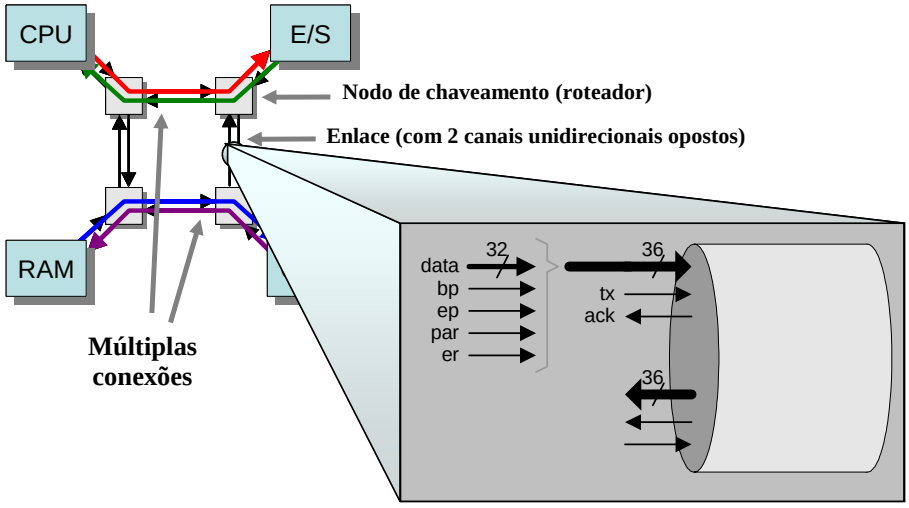


Figura 2.3: Enlaces: ligações físicas entre os nodos da NoC.

Em SoCs os enlaces são implementados através da conexão entre os módulos. É neste quesito que as topologias NoC superam as topologias de barramento. Nas topologias NoC, as conexões são locais, entre módulos de chaveamento próximos, o que reduz o comprimento total de conexões e, por conseqüência, aumenta o desempenho elétrico. Já nas topologias de barramento, as conexões são globais, o que acarreta perda de desempenho devido aos fios longos. Além disso, é possível que haja múltiplas conexões simultâneas, em canais distintos na NoC, como ilustra a Figura 2.3.

2.1.3 Mensagens e Pacotes

As informações trocadas entre um nodo fonte e um nodo destino de uma comunicação são organizadas sob a forma de mensagens. Geralmente, estas mensagens são quebradas e transmitidas em pacotes. Um pacote corresponde à menor unidade de informação que contém detalhes sobre o roteamento e seqüenciamento dos dados.

Em geral, os pacotes são formados por três partes: um cabeçalho (*header*), um corpo de dados (*payload*) e um terminador (*trailer*). O cabeçalho e o terminador formam um envelope ao redor do corpo de dados do pacote. O cabeçalho carrega informações de roteamento e de controle utilizadas pelos nodos de chaveamento para propagar o pacote através da rede, em direção ao seu destino. Já o terminador, carrega informações utilizadas na sinalização do final do pacote e pode conter informações utilizadas para detecção de erros.

Um pacote é constituído por uma seqüência de *flits*. Um *flit* é a menor unidade de dados sobre a qual é realizado o controle de fluxo, podendo ter o tamanho de um *phit* (largura do canal físico de dados), ou até mesmo de um pacote. Um *phit* é definido pelo número de bits de dado transmitidos simultaneamente. Um *phit* pode incluir, além dos bits de dados, sinais de enquadramento e de controle da integridade do dado (ZEFERINO, 2003).

2.1.4 Características de uma Rede de NoC

Uma NoC pode ser caracterizada quanto à sua topologia, suas estratégias de roteamento, controle de fluxo, chaveamento e arbitragem. Estas características são definidas na Tabela 2.1:

Tabela 2.1: Características de uma NoC.

Característica	Definição/Função
Topologia	Define o arranjo dos nodos e enlaces sob a forma de um grafo.
Roteamento	Determina como uma mensagem escolhe um caminho dentro do arranjo dos nodos e canais de comunicação.
Chaveamento	Define como e quando um canal de entrada de um nodo de chaveamento é conectado a um canal de saída selecionado pelo algoritmo de roteamento.
Controle de fluxo	Lida com a alocação de canais e *buffers* para um pacote que trafega na NoC.
Arbitragem	Determina qual canal de entrada pode utilizar um determinado canal de saída do nodo de chaveamento.
Memorização	Define como e onde serão armazenadas as mensagens bloqueadas em um nodo de chaveamento.

As NoCs são estruturadas em camadas que encapsulam funções equivalentes àquelas definidas para os níveis hierárquicos do modelo de referência OSI (*Open System Interconnection*), um padrão internacional de referência proposto pela ISO (*International Organization for Standardization*) (DAY; ZIMMERMMAN, 1983). O objetivo de uma estrutura de pilha de camadas (ou níveis) de protocolos é delimitar e

isolar funções de comunicações em cada nível. Desta forma, cada nível deve ser pensado como um processo, quer implementado por hardware ou software, que se comunica com o processo correspondente na outra máquina. As regras que governam a conversação de um nível "x" qualquer são chamadas de protocolo de nível "x". O modelo da ISO possui sete níveis de protocolos, como pode ser observado na Figura 2.4.

Figura 2.4: Camadas do modelo OSI (MELLO; MÖLLER, 2003).

A arquitetura da rede é formada por níveis, interfaces e protocolos. Cada nível oferece um conjunto de serviços ao nível superior, usando funções realizadas no próprio nível e serviços disponíveis nos níveis inferiores. Os nodos de chaveamento de uma NoC são estruturados em camadas hierárquicas que implementam algumas das funções dos níveis inferiores (físico, enlace, rede) do modelo OSI, descritas abaixo:

- Nível físico: realiza a transferência de dados em nível de bits através de um enlace.

- Nível de enlace: efetua a comunicação em nível de quadros (grupos de bits). Preocupa-se com o enquadramento dos dados e com a transferência desses quadros de forma confiável, realizando o tratamento de erros e o controle do fluxo de transferência de quadros.

- Nível de rede: faz a comunicação em nível de pacotes (grupos de quadros). Responsável pelo empacotamento das mensagens, roteamento dos pacotes entre a origem e o destino da mensagem, controle de congestionamento e contabilização de pacotes transferidos.

2.2 Topologia

Uma NoC pode ser caracterizada pela estrutura de como seus nodos são interligados. Essa estrutura é tipicamente representada por um grafo G(N,C) onde *N* representa o conjunto de nodos (de processamento e/ou de chaveamento) da rede e *C* representa o conjunto de canais de comunicação. Quanto à topologia, as NoCs podem ser agrupadas em duas classes principais, as redes diretas e as redes indiretas (ZEFERINO, 2003), descritas nas seções seguintes.

2.2.1 Redes Diretas

Nas redes diretas, cada nodo de chaveamento possui um nodo de processamento associado, e esse par pode ser visto como um elemento único dentro da máquina, tipicamente referenciado pela palavra nodo, como ilustra a Figura 2.5. Pelo fato de utilizarem nodos de chaveamento tipo roteador, as redes diretas são também chamadas de redes baseadas em roteadores (DUATO; YALAMANCHILI; NI, 1997).

Cada nodo possui ligações ponto-a-ponto diretas para um determinado número de nodos vizinhos. Um pacote transmitido entre dois nodos não-vizinhos deve passar por um ou mais nodos intermediários. Se um pacote recebido por um nodo é destinado a outro nodo dentro da rede, o nodo de chaveamento do primeiro deve repassá-lo para algum dos seus nodos vizinhos para que o pacote avance em direção ao seu destino. Apenas os nodos de chaveamento são envolvidos nessa comunicação, sem interferência dos nodos de processamento dos nodos intermediários. Um algoritmo de roteamento é utilizado pelo nodo de chaveamento a fim de decidir para qual nodo vizinho o pacote deve ser repassado.

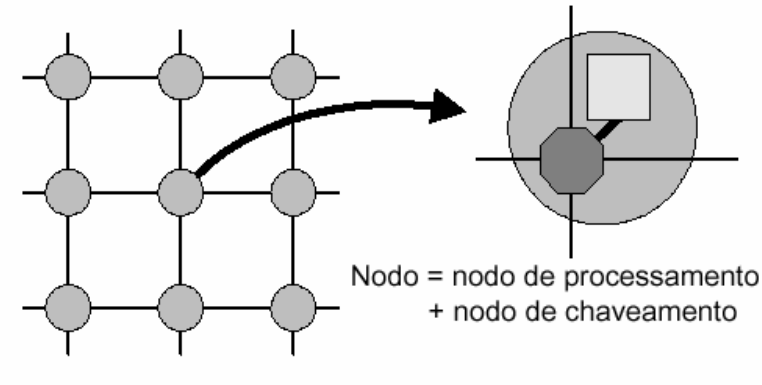


Figura 2.5: Nodos de redes diretas (MELLO; MÖLLER, 2003).

Em termos de conectividade, a rede direta ideal é a completamente conectada. Porém, sua escalabilidade é restrita e seu custo é proibitivo para um número de nodos moderado ou grande, pois o grau do nodo (número de canais por nodo) é *N-1*, onde *N* é o número de nodos da rede (ZEFERINO, 2003).

Devido às dificuldades de se implementar uma rede direta ideal, como a completamente conectada, inúmeras alternativas foram propostas, na tentativa de se obter uma boa relação entre desempenho e custo. A grande maioria das implementações práticas se restringe a alguns modelos de rede que possuem topologia ortogonal. Uma rede apresenta topologia ortogonal se, e somente se, seus nodos podem ser arranjados em um espaço n-dimensional e cada enlace entre nodos vizinhos produz um deslocamento em uma única dimensão. As topologias de redes diretas ortogonais mais utilizadas são a grelha (também chamada de *mesh*) n-dimensional (Figura 2.6a), o toróide (Figura 2.6b) e o hipercubo (Figura 2.6c).

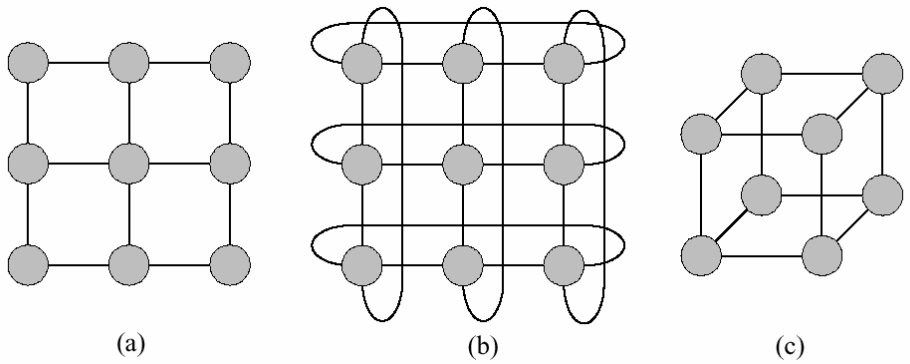


Figura 2.6: (a) Grelha 2D 3x3; (b) Toróide 2D 3x3; (c) Hipercubo 3D (ZEFERINO, 2003).

2.2.2 Redes Indiretas

Nas redes indiretas, o acoplamento entre os nodos de processamento e os nodos de chaveamento não ocorre no mesmo nível das redes diretas. Os nodos de processamento possuem uma interface para uma rede de nodos de chaveamento baseados em chaves. Cada chave possui um conjunto de portas bidirecionais para ligações com outras chaves e/ou com os nodos de processamento. Somente algumas chaves possuem conexões para nodos de processamento e apenas essas podem servir de fonte ou destino de uma mensagem. A topologia da rede é definida pela estrutura de interconexão dessas chaves.

As topologias mais conhecidas de redes indiretas são o *crossbar* e as redes multiestágio. O *crossbar* consiste em roteador com uma chave *N* x *N*, sendo a topologia ideal para a conexão indireta de *N* nodos. Embora seja mais econômico que uma rede direta completamente conectada (a qual necessitaria de *N* roteadores, cada um com uma chave *crossbar N X N*), o *crossbar* possui uma complexidade da ordem de N^2, o que torna o seu custo proibitivo para redes grandes, como por exemplo, 10 x 10.

As redes chamadas de multiestágio são compostas por roteadores usualmente idênticos, organizados como um conjunto de estágios. Nestas topologias, os estágios de entrada e de saída possuem ligações para os nodos e para os estágios internos da rede, que são ligados aos seus vizinhos através de padrões regulares de conexão. Desta forma, uma mensagem tem que atravessar alguns estágios para chegar ao nodo destino.

As redes multiestágio podem ser caracterizadas pelo número de estágios e pela forma como eles são arranjados. A Figura 2.7 (ZEFERINO, 2003) mostra uma rede *crossbar* constituída por um roteador 4 x 4 (quatro portas de entrada e quatro portas de saída) e uma rede multiestágio 8 x 8 bidirecional do tipo borboleta (*butterfly*).

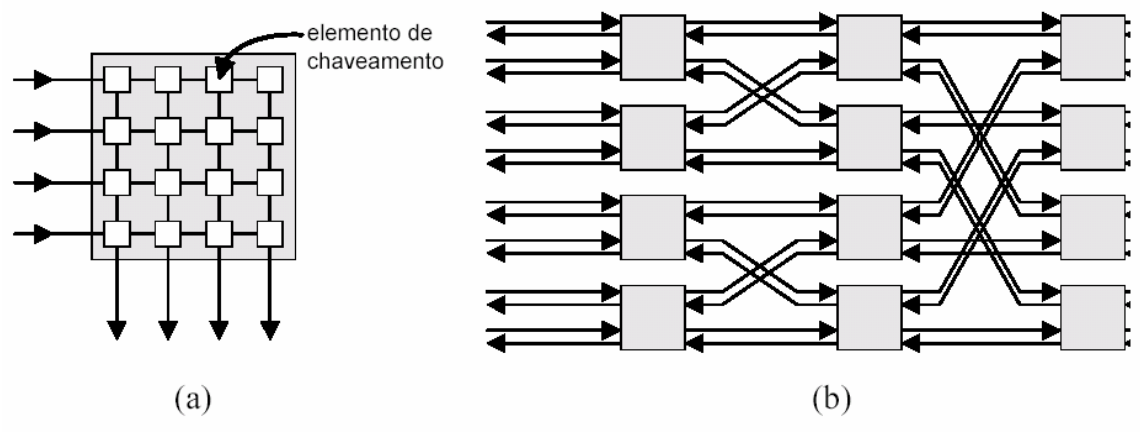


Figura 2.7: Redes indiretas: (a) *crossbar* 4 x 4; (b) multiestágio 8 x 8 bidirecional (ZEFERINO, 2003).

Além das topologias diretas e indiretas apresentadas acima, existem inúmeras outras propostas com objetivos específicos, tais como minimizar o diâmetro de rede para um determinado número de nodos e grau do nodo (FENG, 1981; HWANG, 1993; CULLER; SINGH, 1998). Como exemplo destas topologias podemos citar: o cubo conectado por ciclos (Figura 2.8a), a árvore, a árvore gorda (Figura 2.8b), a estrela, a rede banyan, a banyan-hipercúbica, a pirâmide e a rede De Brujin, entre outras.

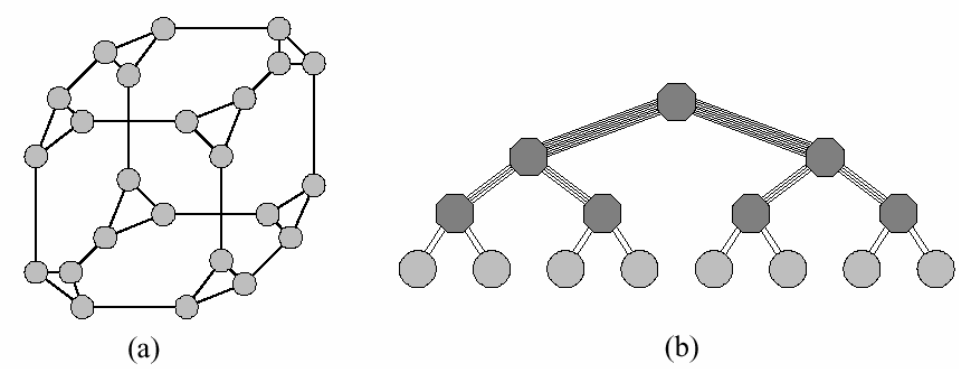


Figura 2.8: (a) Cubo conectado por ciclos 3D; (b) Rede indireta em árvore gorda (ZEFERINO, 1999).

2.3 Roteamento

O roteamento é o método usado por um pacote para escolher um caminho através dos canais e roteadores da rede. O algoritmo de roteamento utilizado tem uma forte influência no desempenho da comunicação na rede. Em geral, o algoritmo de roteamento visa atender a alguns objetivos específicos, os quais têm conseqüência direta em algumas propriedades da NoC, como:

- **Conectividade**: capacidade de rotear pacotes de qualquer nodo fonte para qualquer nodo destino.

- **Liberdade de *deadlock* e *livelock***: capacidade de garantir que nenhum pacote ficará bloqueado ou circulando infinitamente pela rede sem atingir o seu destino. Estes casos serão discutidos na seção 2.8.

- **Adaptatividade**: capacidade de rotear pacotes através de caminhos alternativos quando ocorre congestionamento ou falha em algum componente do caminho em uso.

- **Tolerância a falhas**: capacidade de rotear pacotes na presença de falhas em componentes.

Existem na literatura vários algoritmos de roteamento propostos para atender a requisitos distintos (DUATO; YALAMANCHILI; NI, 1997; ASHRAF; ABD-EL-BARR; AL-TAWIL, 1998). Estes algoritmos podem ser classificados conforme a Tabela 2.2.

Tabela 2.2: Classificação dos algoritmos de roteamento.

Critério	Classificação	Descrição
Quanto ao momento da realização do roteamento:	Dinâmico	O algoritmo de roteamento é realizado no tempo de **execução** da aplicação.
	Estático	O algoritmo de roteamento é realizado no tempo de **compilação** da aplicação.
Quanto ao número de destinos:	*Unicast*	Os pacotes têm um **único** destino.
	Multicast	Os pacotes podem ser roteados para **múltiplos** destinos.
Quanto ao lugar onde as decisões de roteamento são tomadas:	Centralizado	Os caminhos são estabelecidos por um controlador **central**.
	Fonte	O nodo **emissor** (ou fonte) define o caminho a ser seguido pelo pacote antes de injetá-lo na rede.
	Distribuído	O roteamento é realizado pelos **roteadores** enquanto o pacote atravessa a rede.
Quanto à implementação:	Baseado em tabela	O roteamento é feito a partir de uma consulta a uma **tabela** em memória.
	Baseado em máquina de estados	O roteamento é realizado a partir da execução de um **algoritmo** implementado em software ou em hardware.
Quanto à adaptatividade:	Determinístico	O algoritmo de roteamento fornece sempre o **mesmo** caminho entre um determinado par fonte-destino (ou seja, não-adaptativo).
	Adaptativo	O algoritmo de roteamento utiliza alguma informação a respeito do **tráfego** da rede e/ou do estado dos canais para evitar regiões congestionadas ou com falhas.

Fonte: PALMA, 2003.

Os algoritmos adaptativos ainda podem ser sub-classificados quanto à (ao):

- **Progressividade**: progressivo, se o cabeçalho sempre avança pela rede, reservando um novo canal a cada passo de roteamento; ou regressivo (*backtracking*), se o cabeçalho pode retornar pela rede, liberando canais previamente reservados.

- **Minimalidade**: mínimo (ou *profitable*), se o algoritmo de roteamento pode selecionar apenas canais de saída que aproximem cada vez mais o pacote do seu destino; ou não-mínimo (ou *misrouting*), se o algoritmo de roteamento pode selecionar canais que levem o pacote a se afastar do seu destino.

- **Número de caminhos**: completo, se o algoritmo de roteamento pode utilizar todos os caminhos disponíveis; ou parcial, se apenas um subconjunto desses caminhos pode ser usado.

2.4 Chaveamento

Em uma NoC, os dados são transmitidos de uma origem para um destino através de chaves (ou roteadores) intermediárias. Para executar estas transmissões, as chaves devem assumir uma política de repasse de dados para a chave seguinte. As duas políticas de chaveamento utilizadas em NoCs são baseadas ou no estabelecimento de um caminho completo entre o nodo fonte e o destino da mensagem (circuito) ou na divisão das mensagens em pacotes, os quais reservam seus caminhos dinamicamente na medida em que avançam em direção ao seu destino. As seções seguintes apresentam os métodos de chaveamento utilizados em *Networks-on-Chip*.

2.4.1 Chaveamento por Circuito (*Circuit Switching)*

No método de chaveamento por circuito, um caminho completo entre a fonte e o destino da mensagem é estabelecido antes do envio desta mensagem. Este caminho é mantido até o término da transmissão, e qualquer outra requisição de comunicação nos canais alocados é negada. Esta transmissão é feita em duas etapas, como mostra a Figura 2.9. Na primeira, o nodo fonte envia para a rede um cabeçalho de roteamento contendo o endereço do destino e informações de controle. Esse cabeçalho avança pela rede, alocando canais físicos para o estabelecimento do circuito. Se um canal desejado estiver ocupado por outra transmissão, o cabeçalho fica bloqueado até que este canal seja liberado. Quando o cabeçalho atinge o seu destino, uma informação de reconhecimento é enviada para o nodo fonte através do caminho de retorno do circuito estabelecido. Neste momento inicia-se a segunda etapa da comunicação, ou seja, a transferência dos dados da mensagem. O circuito é desfeito através do avanço do terminador da mensagem através dos canais alocados, em direção ao destino. A cada roteador que passa, o terminador sinaliza que o canal pode ser liberado.

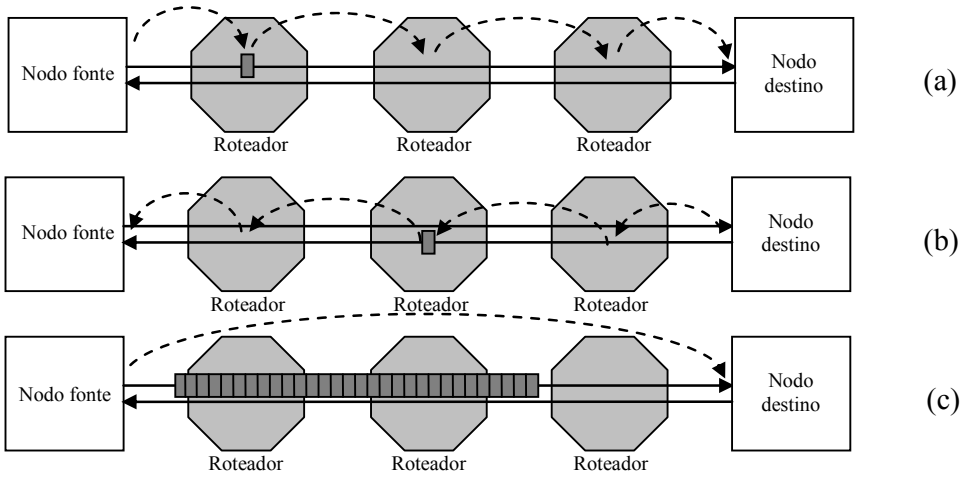


Figura 2.9: Etapas do chaveamento por circuito. Primeira etapa: (a) O cabeçalho avança pela rede, alocando os canais desde sua fonte até seu destino; (b) Uma informação de reconhecimento é enviada para o nodo fonte pelo caminho de retorno do circuito estabelecido. Segunda etapa: (c) Inicia-se a transferência dos dados da mensagem. O terminador desaloca os canais por onde passa (PALMA, 2003).

Este tipo de chaveamento tem sua origem nas redes telefônicas. Sua grande vantagem está no fato de que, uma vez estabelecido o caminho entre os nodos fonte e destino, a mensagem é transferida sem bloqueios. Isto faz com que não seja necessária a utilização de filas nos roteadores, mas sim pequenos *buffers* para armazenar o cabeçalho enquanto um canal desejado estiver ocupado.

A desvantagem deste método é que ele causa degradação do desempenho do sistema como um todo, pois o caminho entre a origem e o destino fica alocado durante toda a fase de transmissão de dados, não podendo ser utilizado por outro circuito. O uso deste método se justifica apenas em casos onde as mensagens são longas, mas pouco freqüentes. Alguns exemplos de máquinas paralelas que utilizaram o chaveamento por circuito incluem o Intel iPSC/2 (NUGENT, 1988), Intel iPSC/860 e o BBN GP 1000.

2.4.2 Chaveamento por Pacote (*Packet Switching)*

Em casos onde as mensagens trocadas entre nodos da rede são curtas e freqüentes, o uso do chaveamento por circuito é injustificável, pois aumenta a contenção na rede e o tempo para estabelecer um circuito torna-se muito maior que o tempo envolvido na transferência das mensagens. Uma alternativa é quebrar as mensagens em pacotes que são transmitidos pela NoC. Cada pacote possui um cabeçalho com as informações necessárias para o seu roteamento. Estes pacotes são transmitidos um a um, sendo que cada pacote reserva apenas os recursos necessários para avançar de nodo em nodo.

A grande vantagem deste método é que o canal permanece ocupado apenas enquanto o pacote está sendo transferido. Sua desvantagem deve-se ao fato de que em cada roteador são necessárias filas para armazenar temporariamente um pacote inteiro ou parte dele, dependendo da técnica utilizada. As três técnicas de chaveamento por pacote mais utilizadas são *store-and-forward*, *virtual-cut-through* e *wormhole* (RIJPKEMA, 2001; MOHAPATRA, 1998).

2.4.2.1 Store-and-forward (SAF)

Neste método, um roteador recebe um pacote, armazena-o em seu buffer, identifica o destino da mensagem, seleciona uma porta de saída com base em algum critério de roteamento e repassa o pacote adiante para um roteador adjacente ou para o nodo de processamento local, caso este seja o destinatário do pacote. Por esse motivo, essa técnica é denominada de *store-and-forward* (SAF), ou seja, armazena-e-repassa.

Pelo fato de alocar somente os recursos necessários para avançar de nodo para nodo na rede, este método implica em uma sobrecarga adicional à comunicação, pois cada um dos pacotes deve transportar um cabeçalho de endereçamento e os roteadores precisam gastar um tempo para efetuar o roteamento individual de cada pacote. Outra desvantagem dá-se pelo fato de que um pacote só é repassado após ter sido completamente recebido, o que aumenta a latência da comunicação de acordo com o tamanho do pacote. Além disso, são necessários buffers em todos os roteadores para manter pacotes inteiros, o que aumenta o custo da rede, como mostra a Figura 2.10.

O chaveamento *store-and-forward* foi inicialmente utilizado em redes de comunicação de computadores. As primeiras máquinas paralelas que utilizaram este tipo de chaveamento foram: Intel iPSC/1, MIT Tagged Dataflow Machine (ARVIND; NIKHIL, 1987) e Manchester Dynamic Dataflow Machine (GURD; KIRKHAM; WATSON, 1985; GURD; SNELLING, 1987).

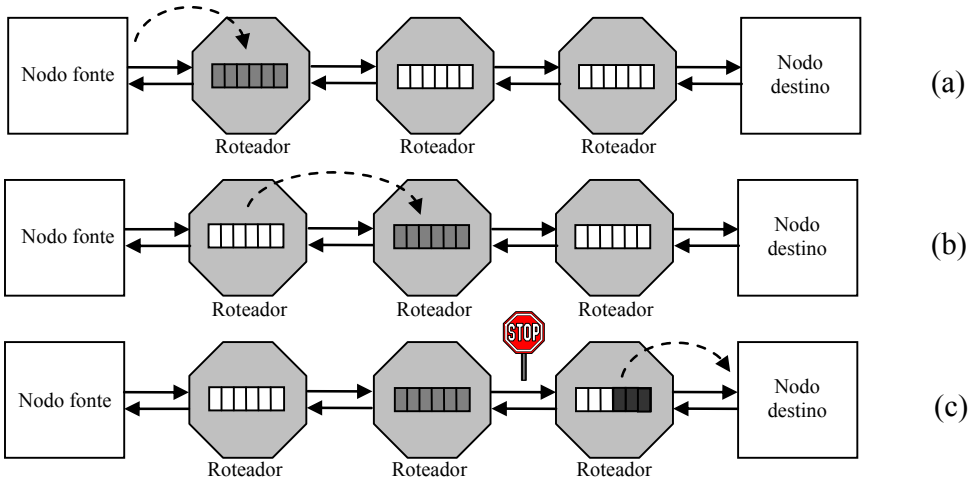


Figura 2.10: Chaveamento SAF: (a) O roteador recebe todo o pacote e, só então, (b) repassa ao próximo roteador. (c) O pacote fica bloqueado, pois o próximo roteador ainda não possui espaço suficiente para armazená-lo completamente, devido a um outro pacote, ainda em transmissão (PALMA, 2003).

2.4.2.2 Virtual-cut-through (VCT)

O método de *virtual-cut-through* (Figura 2.11) foi proposto por Kermani e Kleinrock (KERMANI; KLEINROCK, 1979) como um aperfeiçoamento do método *store-and-forward*. Sua vantagem em relação ao método anterior é que ele só armazena o pacote inteiro no roteador quando o canal que deseja está ocupado. Isto reduz a latência da comunicação quando o canal não está ocupado, pois, neste caso, o pacote é imediatamente repassado, não precisando de armazenamento. Na técnica SAF, mesmo quando o canal está disponível o pacote deve ser armazenado inteiramente para, só então, ser retransmitido.

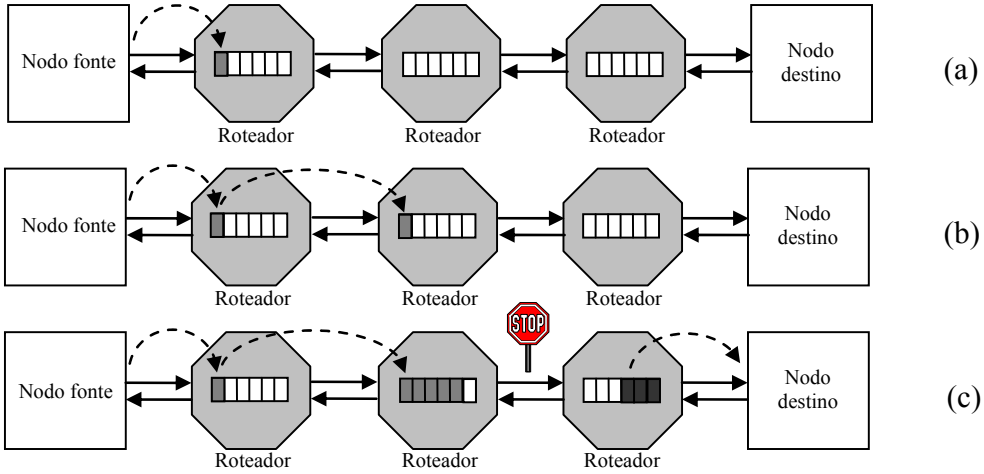


Figura 2.11: Chaveamento VCT: (a) O primeiro roteador recebe um *flit* de cabeçalho, executa o algoritmo de roteamento e (b) o repassa para o próximo roteador, ao mesmo tempo em que o segundo *flit* chega no primeiro roteador. (c) O pacote fica bloqueado no segundo roteador, pois o canal de saída do terceiro roteador está ocupado e este roteador não possui espaço suficiente para armazenar todo o pacote (PALMA, 2003).

A desvantagem do VCT é a necessidade de *buffers* para armazenar todo o pacote (como no SAF), quando o canal está ocupado. No pior caso, quando a rede está muito carregada, o VCT se comporta como o chaveamento SAF. Um exemplo de aplicação do chaveamento VCT é o roteador Chaos (KONSTANTINIDOU; SNYDER, 1991).

2.4.2.3 Wormhole

O chaveamento por *wormhole* foi criado por Dally e Seitz (DALLY; SEITZ, 1986). Trata-se de uma variação do chaveamento VCT que visa reduzir a quantidade de *buffers* necessários para manter pacotes bloqueados na rede. Neste método o pacote é dividido em *flits*, sendo transmitidos entre as chaves intermediárias até o destino. O método *wormhole* funciona como um *pipeline*, onde os *flits* de cabeçalho (contendo informações do destino) se movem pela rede seguidos pelos *flits* de dados (*payload*). Quando os *flits* de cabeçalho são bloqueados, os *flits* do corpo do pacote ocupam as filas das chaves intermediárias. Um pacote deve atravessar completamente um canal antes de liberá-lo para outro pacote. Essa é uma das principais desvantagens dessa técnica de chaveamento, pois a probabilidade de ocorrer o *deadlock* é maior. Entretanto, essa restrição pode ser contornada através do uso de canais virtuais, onde múltiplos canais lógicos compartilham um único canal físico.

A vantagem do *wormhole* é que a latência não depende diretamente da distância, como os métodos anteriores, mas basicamente do tráfego entre as chaves de origem e destino. Outra vantagem deste método é a redução das filas nas chaves intermediárias, que não precisam armazenar o pacote inteiro, o que possibilita a construção de roteadores pequenos e rápidos. A desvantagem do método é a contenção de recursos causada pelo bloqueio do pacote. O chaveamento *wormhole* é o mais usado atualmente, por oferecer mais vantagens com relação à utilização da rede e ao custo dos roteadores.

Como exemplos de máquinas com redes de interconexão baseadas no chaveamento *wormhole* pode-se citar: o Cray T3D (KESSLER; SCHWARZMEIER, 1993), Cray T3E

(SGI, 1999), IBM SP (STUNKEL, 1994), Meiko CS-2 (BEECROFT, 1994), SGI Origin 2000 (LAUDON; LENOSKI, 1997) e Intel/ASCI Option Red (MATTSON; HENRY, 1998).

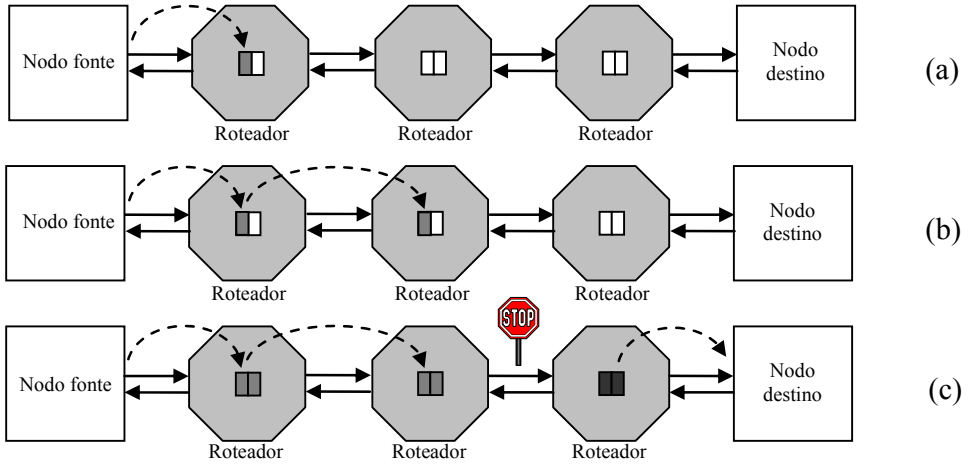


Figura 2.12: Chaveamento *wormhole*: (a) O roteador recebe um *flit* de cabeçalho, executa o algoritmo de roteamento e (b) o repassa para o próximo roteador, ao mesmo tempo em que o segundo *flit* chega no primeiro roteador. (c) Quando o *flit* de cabeçalho é bloqueado, os *flits* do corpo do pacote ocupam as filas das chaves intermediárias (PALMA, 2003).

2.5 Controle de Fluxo

Os pacotes que trafegam por uma NoC competem por seus canais e *buffers* na medida em que avançam em direção aos seus destinos. Quando um pacote necessita de um recurso que já está sendo utilizado por outro pacote, diz-se que houve uma colisão. Quando ocorre uma colisão, é necessário utilizar uma política para decidir se o pacote deve ser descartado, bloqueado onde está, recebido e armazenado temporariamente, ou desviado para um outro caminho. Esta política é chamada de controle de fluxo e lida com a alocação dos canais e dos *buffers* para a transmissão de um pacote pela rede (NI; McKINLEY, 1993).

As NoCs realizam o controle de fluxo em nível de enlace. Geralmente os roteadores possuem, em cada terminal de um enlace, *buffers* para armazenamento dos dados em transferência. Quando o *buffer* de entrada do receptor está cheio, o transmissor deve manter o dado a ser enviado em um *buffer* local até que o receptor esteja apto a recebê-lo. Para isto, é necessário um mecanismo de controle que bloqueie a saída de dados do transmissor e que envie uma informação do receptor ao transmissor indicando se ele está pronto ou não para receber novos dados.

O método mais simples de controle de fluxo consiste na utilização de um *buffer* FIFO de entrada no receptor, e uma linha de retorno ao transmissor para informar se há espaço disponível neste *buffer*. Essa informação pode ser interpretada pelo transmissor como sendo um crédito e ele só envia um dado se tiver crédito disponível.

Uma outra abordagem é o protocolo de *handshake* (aperto de mão), no qual o transmissor informa ao receptor a existência de um dado a ser enviado através de uma linha (*tx*) e o receptor confirma a disponibilidade para receber este dado através de uma

linha de reconhecimento (*acknowledge*). Outros métodos de controle de fluxo incluem o uso de *slack buffers* e de canais virtuais, apresentados nas Seções seguintes.

2.5.1 Controle de Fluxo Baseado em *Slack Buffer*

O controle de fluxo baseado em *slack buffer* (SEITZ; SU, 1993) pode ser comparado com o controle de nível em um reservatório onde seu conteúdo deve ser mantido entre marcas de nível baixo e nível alto (região chamada de histerese). Quando o conteúdo do *buffer* ultrapassa um destes limites, o receptor deve enviar um sinal de controle ao transmissor, fazendo com que este suspenda ou restabeleça o envio de dados de forma a regular o fluxo da transmissão.

Este método oculta a latência associada à transmissão das mensagens de controle de fluxo. Para isto, a região de histerese deve ser dimensionada de forma a evitar o envio excessivo de sinais de controle, pois estes consomem largura de banda no sentido contrário ao do fluxo de dados. Este tipo de controle de fluxo é bastante adequado às redes que utilizam chaveamento *wormhole*, como por exemplo, a rede Myrinet (BODEN; COHEN; FELDERMAN; KULAWIK; SEITZ, 1995).

2.5.2 Controle de Fluxo Baseado em Canais-Virtuais

Os canais-virtuais foram introduzidos por Dally e Seitz (DALLY; SEITZ, 1987) para resolver o problema de *deadlock* em redes com chaveamento *wormhole*. Nestas redes, cada roteador possui em *buffer* (ou fila) na entrada de cada canal físico para armazenar os *flits* recebidos até que os mesmos sejam direcionados a uma porta de saída, de acordo com o roteamento realizado. No caso do primeiro pacote deste *buffer* ser bloqueado devido a uma colisão de recursos, e o pacote ser maior que o espaço disponível no *buffer*, este canal físico também ficará bloqueado e nenhum outro pacote poderá utilizá-lo. Este problema é conhecido como bloqueio de cabeça de linha (HOL – *Head-of-Line blocking*).

O *buffer* de entrada pode ser organizado em filas de profundidade menor, podendo ser alocadas independentemente umas das outras (canais-virtuais). Neste caso, se um canal for bloqueado, o canal físico ainda poderá ser usado por outro canal-virtual, evitando o problema de bloqueio de cabeça de linha e aumentando a utilização do canal físico. Esta organização pode ser chamada de multi-via, pois existem múltiplas vias virtuais em uma mesma via física, enquanto que a organização anterior pode ser chamada de mono-via.

2.5.3 Controle de Fluxo Baseado em Créditos

No controle de fluxo baseado em créditos, uma transmissão só é realizada quando o receptor tem espaço suficiente em seu *buffer* para armazenar o pacote a ser recebido (KORNAROS, G.; PNEVMATIKATOS, D.; VATSOLAKI, P.; KALOKERINOS, G.; XANTHAKI, C.; MAVROIDIS, D.; SERPANOS, D.; KATEVENIS, M., 1999), como já foi mencionado anteriormente. Neste tipo de controle, o receptor envia ao transmissor uma informação indicando o espaço (créditos) disponível no seu *buffer* para recepção. O transmissor inicia a transmissão somente quando houver crédito suficiente, e seu crédito diminui à medida que envia dados e só aumentam quando o transmissor recebe mensagens de controle apropriadas. O controle de fluxo baseado em créditos pode ser utilizado em canais físicos multi-via ou mono-via. Um exemplo de rede de interconexão que utiliza este tipo de controle é a rede do multiprocessador SGI Origin 2000 (GALLES, 1997), onde cada canal físico possui quatro canais-virtuais.

2.6 Memorização

Em redes que utilizam chaveamento por pacote (seja do tipo SAF, VCT ou *wormhole*), os roteadores devem ser capazes de armazenar os pacotes destinados a saídas que estejam sendo utilizadas por outros pacotes e realizar o controle de fluxo para evitar a perda de dados. Para isto, é necessário utilizar um esquema de memorização para manutenção dos pacotes bloqueados no roteador. O caso ideal seria ter roteadores com capacidade de armazenamento infinita e garantir que nenhum pacote fosse bloqueado por outro quando sua saída fosse liberada. Porém, a memória do roteador é limitada e, dependendo da estratégia utilizada, o bloqueio de um pacote é inevitável. A organização dos *buffers* de memória e suas localizações interferem no desempenho do roteador. As seções seguintes apresentam algumas estratégias de memorização que podem ser utilizadas em roteadores.

2.6.1 Memorização Centralizada Compartilhada

Neste caso, utiliza-se um *buffer* centralizado no roteador para armazenar os pacotes bloqueados de todas as portas de entrada. Este *buffer* é denominado CBDA (*Centrally-Buffered, Dynamically-Allocated*), pois seu espaço de endereçamento é dinamicamente distribuído entre os pacotes bloqueados. Sua largura de banda, no pior caso, deve ser igual à soma das larguras de banda de todas as portas, ou seja, em um roteador NxN, o *buffer* deve possuir $2N$ portas de modo a permitir N acessos simultâneos de leitura e N acessos simultâneos de escrita.

Esta abordagem oferece uma utilização do espaço de memória melhor do que aquelas proporcionadas pelas abordagens nas quais esse espaço é previa e estaticamente alocado às portas de entrada. Porém, se uma saída estiver em uso por uma entrada e outra entrada com pacotes destinados a essa saída continuar a receber dados, isto levará ao enchimento do *buffer*, afetando as outras comunicações. Isto pode ser evitado através da limitação do espaço alocável a cada porta. Um exemplo de roteador que utiliza esta memorização centralizada compartilhada é o roteador Vulcan da máquina IBM SP-2 (IBM, 1999-a; IBM, 1999-b).

2.6.2 Memorização na Entrada

Neste caso, são utilizados *buffers* independentes nas portas de entrada do roteador. Existem várias implementações possíveis de memorização na entrada, como por exemplo, as estratégias de *buffer* FIFO, *buffers* SAFC, *buffers* SAMQ e *buffers* DAMQ (TAMIR; FRAZIER, 1992).

2.6.2.1 Buffers FIFO

Nesta estratégia, a mais simples e de menor custo, cada buffer FIFO (First-In, First-Out) possui um espaço de memória fixo onde os dados são lidos na mesma ordem em que são escritos. Seu grande inconveniente é o problema do bloqueio HOL, ou seja, um pacote bloqueado na saída do buffer impede o avanço de outro pacote que esteja atrás dele e para o qual a saída desejada esteja disponível. Isto resulta em uma subutilização das portas de saída. A Figura 2.13 mostra um roteador com quatro buffers FIFO nas entradas e um crossbar 4x4.

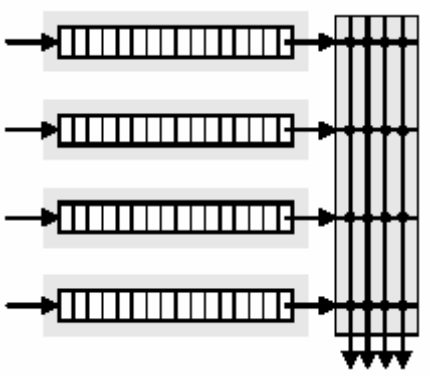


Figura 2.13: Roteador com quatro *buffers* FIFO (TAMIR; FRAZIER, 1992).

2.6.2.2 Buffers SAFC

A estratégia SAFC (*Statically Allocated, Fully Connected*) é uma alternativa para contornar o problema do bloqueio HOL e consiste em dividir cada *buffer* de entrada em *N* partições com tamanho igual a 1/*N* do tamanho do *buffer* original, conforme é mostrado na Figura 2.14a. Esta alternativa requer o uso de um *crossbar* N^2x*N* ou, então, de *N crossbars N*x1 ao invés de um único *crossbar N*x*N*.

A desvantagem do SAFC é o custo adicional referente ao controle do núcleo de chaveamento e ao controle dos *N buffers* por porta de entrada. Além disso, a taxa de utilização dos *buffers* é limitada a 1/*N* do espaço de armazenamento total, ou seja, pior que a dos *buffers* FIFO, e o controle de fluxo é mais complexo, pois deve ser realizado para cada *buffer* de entrada, e exige um pré-roteamento dos pacotes recebidos para que os mesmos sejam direcionados à partição correspondente à saída a ser requisitada (TAMIR; FRAZIER, 1992).

2.6.2.3 Buffers SAMQ

A estratégia SAMQ (*Statically Allocated Multi-Queue*), mostrada na Figura 2.14b visa simplificar o gerenciamento do *crossbar* através da multiplexação das saídas dos *buffers* de entrada atribuídos a uma mesma porta de saída. A estratégia SAMQ elimina alguns dos inconvenientes da SAFC, pois reduz o custo do *crossbar*. Porém, ela ainda mantém os outros dois problemas relacionados à utilização dos *buffers* e ao controle de fluxo.

2.6.2.4 Buffers DAMQ

A estratégia DAMQ (*Dynamically-Allocated, Multi-Queue*) foi proposta por Tamir (TAMIR; FRAZIER, 1992) com o objetivo de evitar os problemas das estratégias anteriores. Na DAMQ, o espaço no *buffer* é associado a uma porta de entrada e particionado dinamicamente entre as portas de saída conforme a demanda dos pacotes recebidos (Figura 2.14c). Assim, é possível evitar o bloqueio HOL dos *buffers* FIFO e aumentar a utilização do espaço de memória disponível. Outra vantagem é que o controle de fluxo na DAMQ é mais simples que nas estratégias SAFC e SAMQ, não requerendo pré-roteamento. Sua desvantagem é a implementação física complexa do gerenciamento do *buffer* DAMQ, baseado em listas encadeadas.

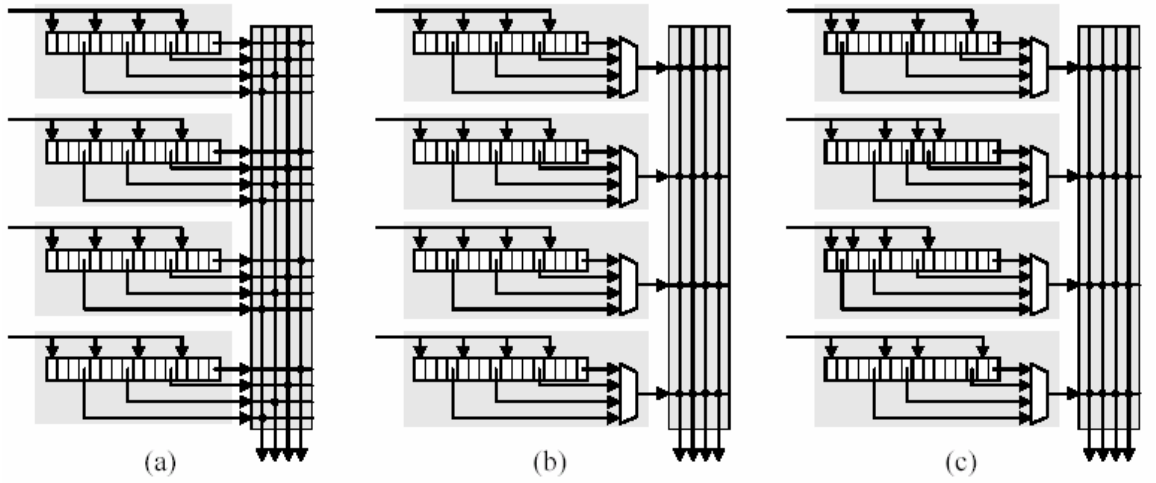


Figura 2.14: Roteador com quatro *buffers*: (a) SAFC; (b) SAMQ; (c) DAMQ (TAMIR; FRAZIER, 1992).

2.6.3 Memorização na Saída

Nesta abordagem, o espaço de memorização é particionado entre as saídas, e estas partições podem ser implementadas como *buffers* FIFO. Cada *buffer* deve ser capaz de suportar a demanda simultânea das *N* entradas, podendo ser implementado com *N* portas de escrita ou com uma porta de escrita operando a uma velocidade *N* vezes maior que a das entradas. Uma dificuldade na implementação desta abordagem é que ela requer um controle de fluxo interno entre portas de entrada e de saída do roteador.

2.7 Arbitragem

A arbitragem é responsável por definir qual porta de entrada (ou *buffer* de entrada) poderá utilizar uma determinada porta de saída (ou *buffer* de saída) em um determinado momento. Este mecanismo é essencial para resolver conflitos causados pela existência de múltiplos pacotes competindo por uma mesma porta de saída. Ele deve ser capaz de resolver esses conflitos, selecionando um dos pacotes com base em algum critério e sem levar qualquer pacote a sofrer *starvation*, ou seja, ficar indefinidamente esperando por uma oportunidade para avançar em direção ao nodo destino.

O árbitro de um roteador pode ser implementado de duas formas: centralizada ou distribuída. Na forma centralizada, mostrada na Figura 2.15a (ZEFERINO, 2003), os mecanismos de roteamento e de arbitragem são implementados em um único módulo que recebe os cabeçalhos dos pacotes, executa o roteamento e determina a porta de saída a ser utilizada por cada pacote. A seguir, é feita a arbitragem, onde o módulo seleciona os pacotes a serem conectados em cada saída, configura o *crossbar* e habilita os *buffers* selecionados para os mesmos transmitirem seus pacotes.

Os árbitros que utilizam esta abordagem visam maximizar a utilização do *crossbar*, realizando uma arbitragem global. Isto por que ela leva em consideração todos os pacotes que estejam prontos para serem transmitidos nos *buffers* das portas de entrada, bem como o estado atual das portas de saída. A desvantagem da arbitragem centralizada é que ela impõe maiores restrições quanto à capacidade de roteamento de pacotes no tempo.

Na forma distribuída (Figura 2.15b (ZEFERINO, 2003)), o roteamento e a arbitragem são realizados de forma independente para cada porta do roteador. Em cada

porta do roteador existe um módulo de roteamento associado a ela, juntamente com um módulo de arbitragem na sua porta de saída (sendo que as portas são bidirecionais). Desta forma, aumenta-se a capacidade de roteamento de pacotes no roteador. Entretanto, existem algumas limitações quanto à sua aplicação em redes baseadas em algoritmos de roteamento adaptativo. Isto acontece porque o mecanismo de roteamento pode determinar múltiplas saídas candidatas e então enviar requisições a todos os árbitros selecionados (CULLER; GUPTA; SINGH, 1997). Se pelo menos dois dos árbitros requisitados selecionarem a mesma entrada, uma saída terá que ser escolhida e as outras ficarão ociosas, diminuindo a taxa de utilização do *crossbar*. Este problema é resolvido se o árbitro tiver uma visão global, como na forma de implementação centralizada. A forma distribuída é mais adequada a redes que utilizam roteamento determinístico.

Existem alguns critérios nos quais os mecanismos de arbitragem podem ser baseados, como por exemplo, esquemas com prioridades estáticas, prioridades dinâmicas, escalonamento por idade (ou *deadline*), FCFS (*First-Come-First-Served*), LRS (*Least Recently Served*) e RR (*Round-Robin*). Dentre estes esquemas, o mais simples de ser implementado é o de prioridades estáticas, constituindo um circuito codificador de prioridade simples, Nele, cada requisição considerada pelo árbitro tem um nível de prioridade fixo. Desta forma, dependendo do tráfego da rede, uma requisição com menor prioridade pode ficar sempre esperando para ser atendida, vindo a sofrer *starvation*. A solução para este problema é a utilização de um esquema de prioridades dinâmicas como, por exemplo, um mecanismo implementado através de uma fila circular (ou *Round-Robin*) baseado em um codificador de prioridade programável.

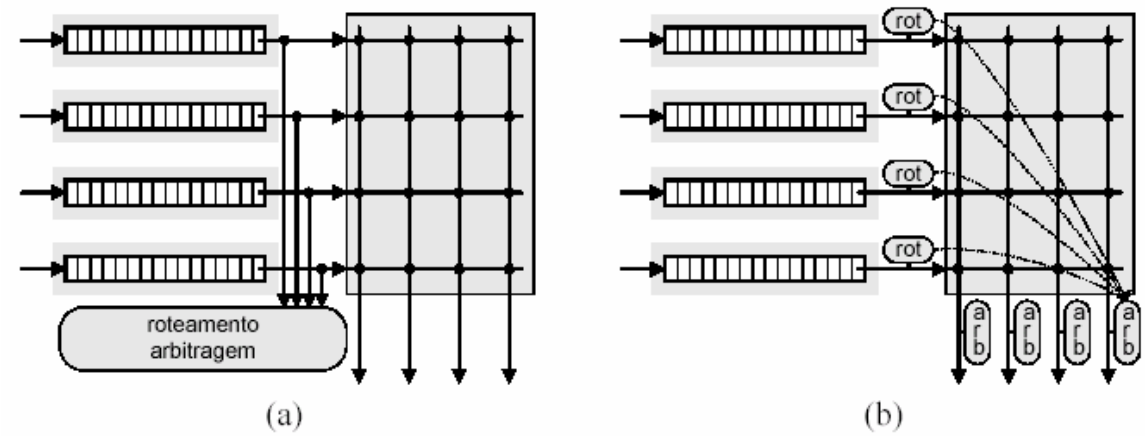


Figura 2.15: Formas para implementação de árbitros em roteadores: (a) centralizada; (b) distribuída (ZEFERINO, 2003).

2.8 Starvation, Livelock e Deadlock

Como visto anteriormente, em uma NoC, os pacotes trafegam através dos canais físicos e dos *buffers* dos nodos de chaveamento. Uma comunicação só é realizada com sucesso quando a informação enviada chega ao seu destino. Porém, existem três casos que devem ser evitados, pois impedem que a comunicação seja realizada: *starvation*, *livelock* e *deadlock*. Estes casos são descritos nas seções seguintes.

2.8.1 Starvation

Quando dois ou mais *buffers* de entrada de um nodo de chaveamento possuem pacotes destinados a uma mesma saída, é necessário que o mecanismo de arbitragem escolha qual destes *buffers* deve ser conectado à saída. Dependendo dos critérios de arbitragem utilizados, um pacote com baixa prioridade pode ficar bloqueado permanentemente, esperando por um recurso que é sempre concedido a outros pacotes de maior prioridade. Este caso, chamado de *starvation*, pode ser evitado através do uso de mecanismos de arbitragem adequados. Se pacotes com prioridade mais alta que os outros ocupam toda a banda, a solução é reservar uma parte desta banda para os pacotes com prioridades mais baixas.

2.8.2 Livelock

Outro caso é o chamado de *livelock*, que ocorre quando um pacote trafega permanentemente pela rede porque os canais necessários para que ele atinja seu destino nunca se encontram disponíveis. Este problema acontece, geralmente, em algoritmos de roteamento tolerantes a falhas, pois estes utilizam caminhos não-mínimos para rotear os pacotes. A forma mais simples de evitar este problema é utilizar algoritmos de roteamento que permitam apenas caminhos mínimos, ou seja, os caminhos mais curtos até o destino. Porém, quando o uso de caminhos não-mínimos se faz necessário, como para oferecer tolerância a falhas, pode-se prevenir o *livelock* através da limitação do número de operações de desvio através de caminhos não-mínimos.

2.8.3 Deadlock

O terceiro caso onde um pacote pode não atingir seu destino é o chamado *deadlock*. Este é o problema mais difícil de ser resolvido, e ocorre quando existe uma dependência cíclica de recursos na rede, como mostra a Figura 2.16 (ZEFERINO, 2003).

A Figura 2.16a mostra um modelo de roteador com quatro portas bidirecionais (1, 2, 3 e 4) conectadas a um núcleo *crossbar* (X). A Figura 2.16b mostra a dependência cíclica em uma parte de uma NoC com quatro nodos de chaveamento (A, B, C e D) interligados através dos enlaces Cab, Cbc, Ccd e Cda. Esta dependência acontece porque existem quatro pacotes neste segmento de rede, cada um deles mantendo um canal de enlace e esperando para utilizar outro canal já alocado para outro pacote. É o caso do pacote no enlace Cab (e no *buffer* a ele associado), que necessita do canal Cbc para avançar. Porém este canal já está alocado por outro pacote que se encontra bloqueado neste canal. Esta dependência cíclica é ilustrada na Figura 2.16c.

Quando se trata de *deadlock*, três estratégias diferentes podem ser adotadas: (*i*) prevenir, (*ii*) evitar e (*iii*) recuperar.

Se a estratégia adotada for a de prevenção, os recursos (canais e *buffers*) devem ser garantidos a um pacote de modo que uma requisição nunca leve a rede ao *deadlock*. Para isto, é necessário reservar todos os recursos necessários antes do início da transmissão do pacote.

A estratégia de evitar o *deadlock* permite que os recursos sejam alocados na medida em que o pacote avança pela rede. Porém, o recurso só é alocado a um pacote se isto não levar a um estado global inseguro. Uma forma de evitar o *deadlock* é limitar as voltas permitidas pelo algoritmo de roteamento, impedindo a ocorrência de ciclos, como mostrado na Figura 2.16c.

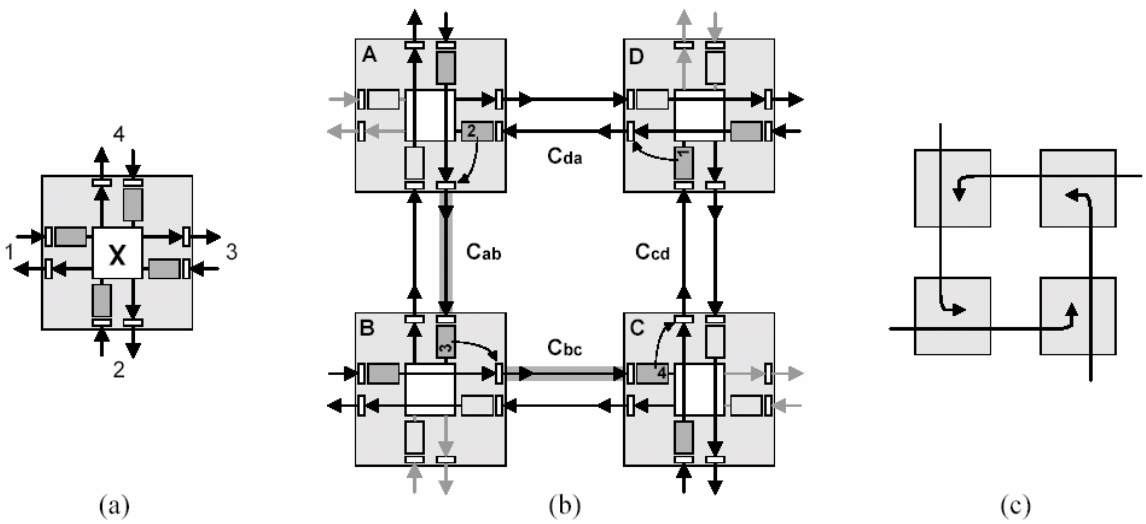


Figura 2.16: *Deadlock:* (a) roteador; (b) pacotes em *deadlock;* (c) dependência cíclica (ZEFERINO, 2003).

Na estratégia de recuperação de *deadlock*, qualquer recurso é garantido a um pacote sem nenhuma verificação, o que pode levar ao *deadlock*. Logo, são necessários mecanismos para detectá-lo e contorná-lo, realocando alguns recursos entre os pacotes. Desta forma, os pacotes que pedem recursos já alocados a eles precisam ser descartados ou roteados novamente. Esta estratégia é conservadora e conduz a uma baixa utilização dos recursos da rede. A estratégia de evitar o *deadlock* é menos conservadora, alocando os recursos na medida em que os pacotes avançam, o que melhora a utilização da rede. Já a estratégia de detecção de *deadlock* é otimista e deve ser utilizada quando os *deadlocks* são raros e suas conseqüências podem ser toleradas (DUATO; YALAMANCHILI; NI, 1997).

2.9 Rede HERMES

A rede Hermes é caracterizada por um bloco construtor básico, o roteador Hermes, ilustrado na Figura 2.17. Este consiste de uma lógica de controle centralizada e de cinco portas bi-direcionais: *East*, *West*, *North*, *South*, and *Local*. Cada porta possui um buffer de entrada para armazenamento temporário de dados. A porta *Local* estabelece uma comunicação entre o roteador e seu núcleo local, enquanto que as demais portas são conectadas a roteadores vizinhos. A lógica de controle implementa o algoritmo de roteamento e o método de arbitragem.

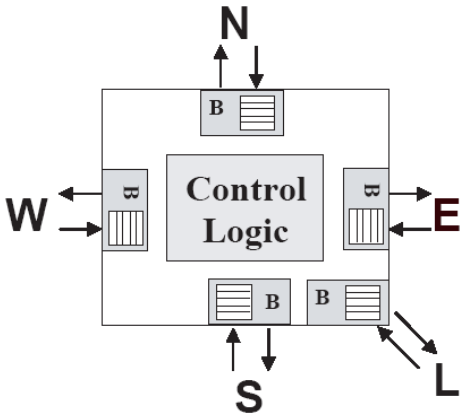


Figura 2.17: Arquitetura básica do roteador Hermes. B indica buffers de entrada (MELLO; MÖLLER, 2003).

A seguir, são apresentadas as características principais da rede Hermes.

- **Topologia**

A NoC Hermes assume que cada roteador possui um conjunto de portas bi-direcionais conectadas a outros roteadores e a um núcleo local. Na topologia *mesh* utilizada neste trabalho, cada roteador possui um número diferente de portas, dependendo de sua posição em consideração aos limites da NoC. Por exemplo, vamos considerar a NoC 3x3 ilustrada na Figura 2.18. Nesta figura, a letra "C" identifica o núcleo local (*IP Core*) conectado à porta local de cada roteador. Os números dentro dos roteadores indicam os seus endereços na NoC, ou seja, as coordenadas X e Y. Nesta NoC, o roteador central, com endereço "11", possui todas as cinco portas ilustradas na Figura 2.17. Entretanto, os roteadores "10", "01", "21" e "12" possuem quatro portas. Já os roteadores "00", "20", "02" e "22", localizados nos cantos, possuem apenas três portas.

A utilização da topologia *mesh* é justificada por facilitar o posicionamento dos roteadores e núcleos no layout, bem como o roteamento dos canais entre roteadores, além de simplificar o algoritmo de roteamento implementado na lógica de controle. O roteador Hermes também pode ser utilizado para construir as topologias toróide, hipercubo e outras topologias similares. Porém, a construção destas outras topologias implica na mudança das conexões entre roteadores e, principalmente, no algoritmo de roteamento.

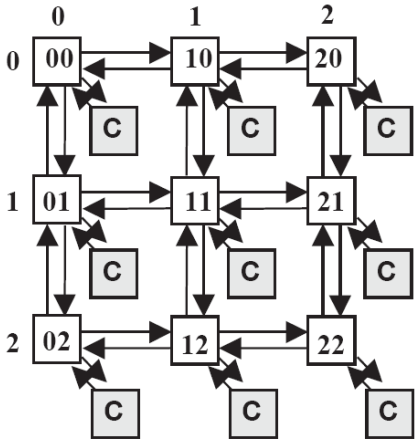


Figura 2.18: Estrutura de uma NoC 3x3 com topologia *mesh*. C indica o IP Core local. Os números dentro dos roteadores indicam os seus endereços na NoC, ou seja, as coordenadas X e Y.

- **Chaveamento**

Dentre os métodos de chaveamento apresentados na seção 2.4, o método *wormhole* foi escolhido por oferecer como vantagens: (*i*) necessita de buffers menores para memorização dos dados; (*ii*) fornece baixa latência na comunicação; e (*iii*) pode multiplexar mais de um canal virtual em um canal físico. Embora, a multiplexação de canais virtuais possa aumentar o desempenho da comunicação (RIJPKEMA, 2001), ela não foi implementada na versão da NoC Hermes utilizada neste trabalho. A razão para isto é a baixa complexidade e custo dos roteadores usando somente um canal virtual em cada canal físico.

Como descrito anteriormente, o chavemanto por *wormhole* divide o pacote em *flits*. Cada pacote possui um cabeçalho de 2 *flits* com as informações necessárias para o seu roteamento. O primeiro *flit* contém o endereço de destino, ou seja, as coordenadas X e Y do roteador de destino, onde X é a posição horizontal e Y a posição vertical. O segundo *flit* indica a quantidade de *flits* que formam o *payload* (área de dados) do pacote. O tamanho do *flit* é parametrizável na rede Hermes, sendo a quantidade máxima de *flits* de *payload* em um pacote limitado em $2^{(\text{largura de } flit, \text{ em bits})}$. Por exemplo, em uma NoC com largura de *flit* igual a 8 bits, o tamanho máximo para o *payload* é de $2^8 = 256$ *flits*.

- **Roteamento**

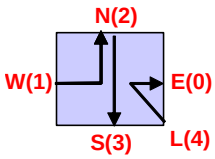
A lógica de roteamento implementa diferentes algoritmos: XY, *West-first minimal*, *North-last minimal* e *Negative-first minimal* (GLASS; NI, 1994). O algoritmo XY é determinístico, roteando o pacote primeiro na direção X e depois na direção Y. Os outros algoritmos são parcialmente adaptativos.

A NoC utilizada neste trabalho foi configurada para utilizar o algoritmo XY. Este algoritmo faz uma comparação entre endereço do roteador atual e o endereço do roteador de destino (armazenado no primeiro *flit* do pacote). O pacote deve ser roteado para a porta local do roteador quando o endereço xLyL (endereço do roteador atual, onde xL é a coordenada horizontal e yL a coordenada vertical) for igual ao endereço xTyT (endereço do roteador de destino do pacote, onde xT é a coordenada horizontal e yT a coordenada vertical). Caso contrário é realizada a comparação horizonal dos

endereços. Quando xL < xT, o pacote dever ser roteado para a porta leste; quando xL > xT, o pacote deve ser roteado para a porta oeste; quando xL = xT, significa que o pacote já está alinhado horizontalmente com o roteador de destino. Neste caso, é realizada a comparação vertical dos endereços. Quando yL < yT, o pacote dever ser roteado para a porta sul; quando yL > yT, o pacote deve ser roteado para a porta norte; quando yL = yT, significa que o pacote já está alinhado verticalmente com o roteador de destino. Neste caso, o pacote deve ser roteado para a porta local do roteador, atingindo o núcleo de destino.

Caso a porta de saída escolhida estiver ocupada, todos os *flits* subseqüentes deste pacote ficam bloqueados, e a requisição permanece ativa até que a conexão com esta porta seja estabelecida. Quando a porta estiver livre, a conexão entre a porta de entrada e a porta de saída é estabelecida. Neste momento, os vetores *in*, *out* e *free* da tabela de chaveamento são atualizados. O vetor *in* indica para qual porta de saída está sendo roteado o pacote de cada porta de entrada. O vetor *out* indica de qual porta de entrada está vindo o pacote sendo roteado para cada porta de saída. O vetor *free* indica se a porta de saída está livre (‘1’) ou ocupada (‘0’). Considerando o exemplo da Figura 2.19, a porta *North* está ocupada (*free* = ’0’), pois está transmitindo na sua porta de saída um pacote recebido da porta *West* (*out* = ‘1’), e a entrada desta porta está recebendo um pacote que está sendo enviado para a porta de saída *South* (*in* = ‘3’). Esta figura apresenta ainda mais duas conexões simultâneas, bem como a tabela de chaveamento para as três conexões. A tabela de chaveamento contém informações redundantes sobre as conexões, porém, esta organização é útil para aumentar a eficiência do algoritmo de roteamento. Ou seja, se a porta *North* está transmitindo um pacote vindo da porta *West* (*out* = ‘1’), fica óbvio que a porta está ocupada, não sendo necessário indicar isto no vetor *free* (*free* = ’0’). Porém, para facilitar a comparação na hora de verificar se a porta está ou não ocupada e, consequentemente reduzir o tamanho da lógica, é mais adequado utilizar um vetor que utiliza apenas 1 bit para cada porta (vetor *free*) indicando sua utilização.

Após todos os *flits* do pacote serem transmitidos pela porta de saída correspondente, a conexão é finalizada, e a posição correspondente a esta porta no vetor *free* volta a ser igual a ‘1’.



(a)

	0 – E	1 – W	2 – N	3 – S	4 – L
Free	0	1	0	0	1
In	-	2	3	-	0
Out	4	-	1	2	-

(b)

Figura 2.19: (a) Três comunicações simultâneas no roteador; (b) Tabela de chaveamento.

- **Controle de fluxo**

É possível escolher entre dois tipos controle de fluxo: *handshake* e *credit-based*. O método *credit-based* consiste na utilização de um *buffer* FIFO de entrada no receptor, e uma linha de retorno ao transmissor para informar se há espaço disponível neste *buffer*. Essa informação pode ser interpretada pelo transmissor como sendo um crédito e ele só envia um dado se tiver crédito disponível.

No método *handshake* (aperto de mão) o transmissor informa ao receptor a existência de um dado a ser enviado através de uma linha (*tx*) e o receptor confirma a disponibilidade para receber este dado através de uma linha de reconhecimento (*acknowledge*).

O método escolhido para a implementação da NoC Hermes utilizada neste trabalho foi o método *credit-based* que apresenta uma latência menor na comunicação.

- **Arbitragem**

O roteador Hermes pode estabelecer até cinco conexões simultâneas. Desta forma, existe a necessidade de um árbitro para determinar qual o pacote deve ser roteado quando mais de um cabeçalho chega ao roteador em um mesmo instante de tempo. O módulo que realiza a arbitragem é apresentado na Figura 2.20.

O árbitro do reoteador Hermes utiliza uma política de arbitragem dinâmica rotativa que permite o roteamento do pacote da porta de entrada com maior prioridade. A prioridade de cada porta depende da última porta que obteve a permissão de roteamento. Por exemplo, se a porta de entrada *Local* (índice 4) foi a última a ter permissão de roteamento, a porta *East* (índice 0) terá a maior prioridade, seguida das portas *West*, *North*, *South* e *Local*, que recebem prioridades decrescentes.

Em outras palavras, a arbitragem é centralizada, utilizando um mecanismo implementado através de uma fila circular (ou *Round-Robin*) baseado em um codificador de prioridade programável. Este método garante que todas as requisições de entrada sejam atendidas, evitando que ocorra *starvation*. A lógica de arbitragem demora quatro ciclos de *clock* para tratar uma requisição de roteamento. Este tempo é necessário para que o algoritmo de roteamento seja executado. Somente após este período o árbitro volta a atender solicitações. Se o algoritmo de roteamento não consegue estabelecer uma conexão com a porta de saída desejada, a porta de entrada volta a solicitar o roteamento ao árbitro, porém com uma prioridade menor.

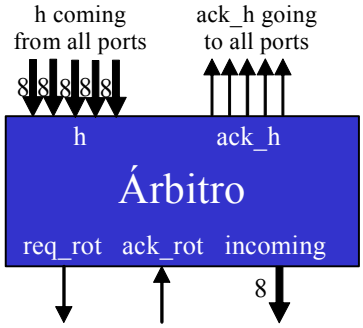


Figura 2.20: Árbitro do roteador Hermes.

A rede Hermes pode ser gerada automaticamente através da ferramenta MAIA (MELLO; OST; PALMA; MORAES; CALAZANS, 2005), incorporada ao ambiente ATLAS, apresentado na seção 2.9.1.

2.9.1 Ambiente Atlas

O ambiente ATLAS automatiza os vários processos relacionados ao fluxo de projeto da rede Hermes. Atualmente, o fluxo de projeto é composto pelas seguintes etapas:

geração da NoC, geração do tráfego, simulação e avaliação de desempenho. Na etapa de geração da NoC, os parâmetros desta, tais como a largura do *flit*, a profundidade dos *buffers*, a quantidade de canais virtuais e a estratégia de controle de fluxo são configurados. Na etapa de geração de tráfego, são gerados cenários de tráfego caracterizando a aplicação a ser executada na NoC. Na etapa de simulação os dados de tráfego são injetados na NoC, havendo, nesta etapa, a transmissão de pacotes através da mesma. Na etapa de avaliação de desempenho, é possível gerar gráficos, tabelas e relatórios para facilitar a análise dos resultados.

A Figura 2.21 mostra a interface gráfica da janela principal do ambiente ATLAS. Esta interface permite que o usuário execute as ferramentas que compõem as etapas do fluxo de projeto.

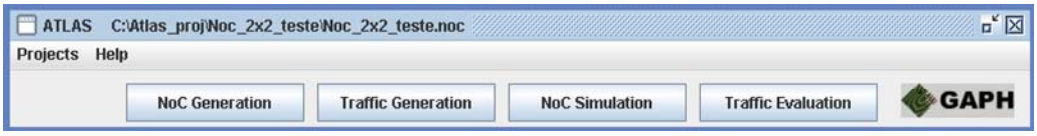


Figura 2.21: Interface gráfica da versão 1.0 do ambiente Atlas.

O ambiente ATLAS é composto por quatro ferramentas:

- **Geração da NoC – Ferramenta MAIA**

A ferramenta MAIA gera a NoC de acordo com as configurações dos parâmetros definidos pelo usuário. A Figura 2.22 apresenta a interface gráfica da ferramenta MAIA em sua versão 4.0. Nesta versão o usuário pode configurar os seguintes parâmetros:

1- Estratégia de **Controle de Fluxo**: *handshake* ou *credit base*;

2- Quantidade de **Canais Virtuais** por canal físico: 1, 2 ou 4;

3- Mecanismo de **Arbitragem**: *Round-Robin* ou *Priority*. A opção *Priority* só pode ser escolhida quando a quantidade de canais virtuais é igual 2 ou 4;

4- **Dimensões** da NoC: de 2 a 16 em ambas as direções, X e Y;

5- **Largura de *flits***: 8, 16, 32 ou 64 bits;

6- **Profundidade dos buffers**: 4, 8, 16 ou 32 *flits*;

7- Algoritmo de **Roteamento**;

8- Geração de ***Testbench*** em SystemC;

9- Botão de **Geração** da NoC;

10- Área de **Visualização** da estrutura da NoC.

A NoC gerada é descrita em VHDL.

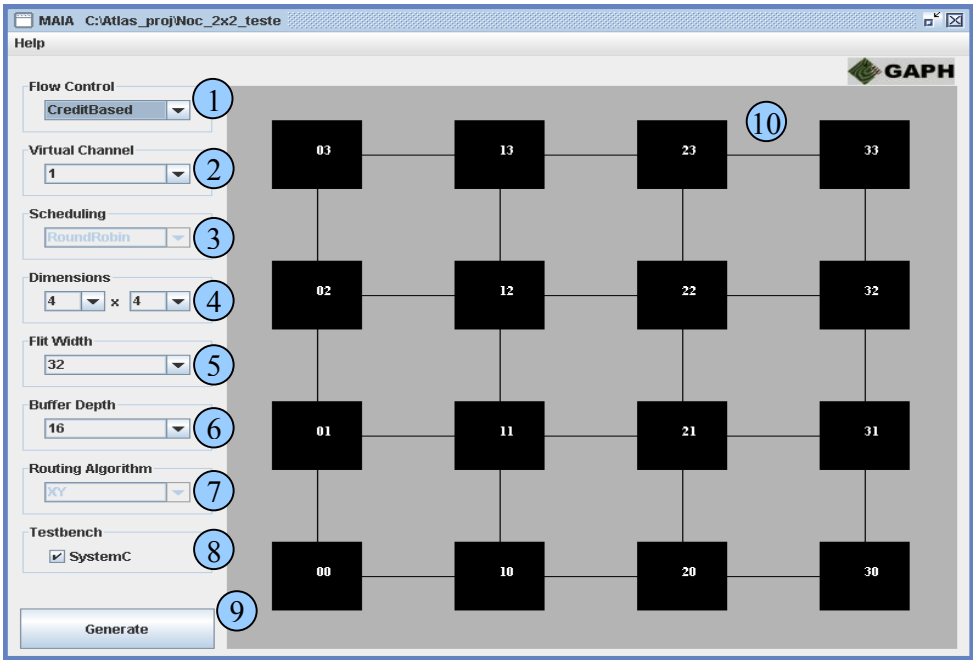


Figura 2.22: Interface gráfica da versão 4.0 da ferramenta MAIA.

Conforme comentado na seção 2.9, os roteadores possuem um número diferente de portas, dependendo de sua posição em consideração aos limites da NoC. Desta forma, os *buffers* de entrada localizados nas portas não utilizadas podem ser removidos destes roteadores, sem afetar o seu funcionamento. Um das vantagens da ferramenta MAIA é o fato de que ela remove estes *buffers* automaticamente, contribuindo para a redução da área e, consequentemente, do consumo de potência em todos os roteadores localizados na periferia da rede Hermes.

- **Geração do Tráfego**

Esta ferramenta produz os arquivos de tráfego que descrevem o conteúdo dos pacotes transmitidos através da NoC. A Figura 2.23 ilustra a interface gráfica da janela principal da ferramenta de geração de tráfego. A opção "*Configuration*" no menu desta janela permite que o usuário defina uma configuração padrão para o tráfego em todos os roteadores da NoC. Entretanto, esta configuração pode ser modificada individualmente para cada roteador. Para isto, basta clicar sobre a imagem do roteador na área de visualização da NoC na interface da ferramenta, que uma janela de configuração será aberta. A Figura 2.24 mostra três configurações possíveis para a distribuição do tráfego: "uniforme", "normal" e "pareto ON/OFF". Os seis parâmetros apresentados no topo destas janelas são independentes. Já os parâmetros na área abaixo na janela, mudam dependendo da escolha do tipo de distribuição selecionado.

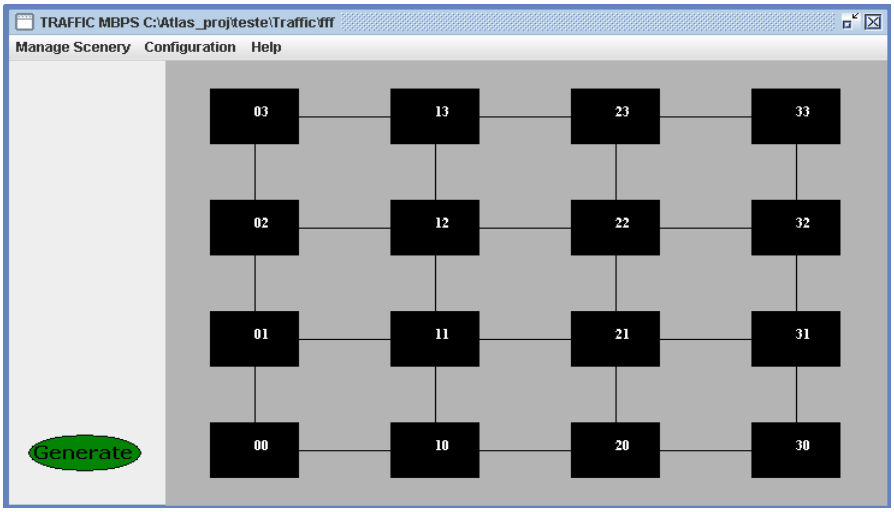


Figura 2.23: Interface gráfica da ferramenta de geração de tráfego.

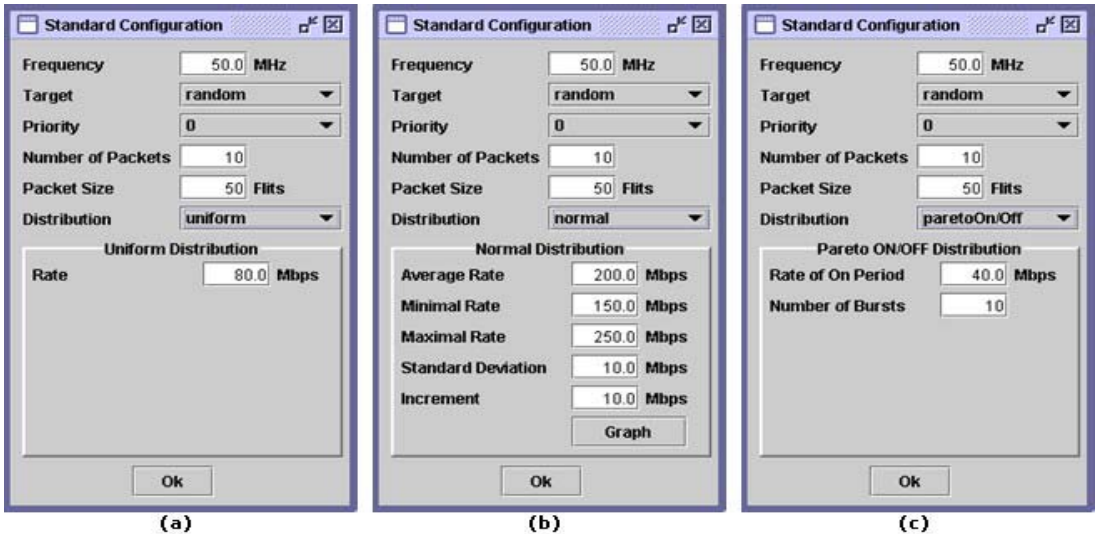


Figura 2.24: Configuração da distribuição do tráfego: (a) uniforme, (b) normal e (c) pareto ON/OFF.

- **Simulação da NoC – Ferramenta *ModelSim***

A ferramenta de simulação da NoC executa um simulador VHDL externo, o *ModelSim*, utilizando como entrada para este simulador a descrição da NoC, juntamente com os *testbenchs* (obtidos na etapa de geração da NoC) e com o padrão de tráfego (gerado na etapa de geração de tráfego). Os resultados da simulação são armazenados em arquivos de saída.

- **Avaliação do desempenho**

A ferramenta de avaliação de desempenho captura os resultados da simulação e permite a análise de vários parâmetros básicos. Isto inclui, mas não se limita, à avaliação do *throughput* e da latência da transmissão em pontos específicos da NoC ou de suas interfaces externas. Para visualizar estes resultados, é possível gerar,

automaticamente, diferentes tipos de gráficos, tabelas e relatórios permitindo uma avaliação detalhada dos mesmos.

2.10 Conclusão

Este capítulo apresentou uma revisão geral sobre os conceitos básicos de *Networks-on-Chip*, abordando sua topologia, algoritmo de roteamento, método de chaveamento, controle de fluxo, estratégia de memorização e mecanismo de arbitragem, além de apresentar três casos que impedem que a comunicação seja realizada e que, portanto, devem ser evitados: *starvation*, *livelock* e *deadlock*. Também foram diferenciados os nodos de chaveamento e os nodos de roteamento, assim como os enlaces, as mensagens e os pacotes. Ao final do capítulo foram apresentas as características da rede Hermes, utilizada como estudo de caso neste trabalho.

O capítulo a seguir trata de diferentes esquemas de codificação de dados encontrados na literatura, alguns deles implementados no escopo deste trabalho, bem como o esquema *T-Bus-Invert*, uma das contribuições deste trabalho.

3 ESQUEMAS DE CODIFICAÇÃO

Quando se utiliza tecnologia CMOS, a potência dissipada nas linhas de comunicação é proporcional ao produto do número médio de transições nestas linhas pela capacitância da linha. Desta forma, uma maneira de reduzir a dissipação de potência na comunicação é codificar o dado a ser transmitido, através de esquemas de codificação que reduzam o número médio de transições nestas linhas.

Com base nesta observação, muitos esquemas de codificação foram propostos nos últimos anos, tendo como objetivo reduzir o consumo de potência em barramentos. Alguns destes esquemas (STAN; BURLESON, 1995; BENINI et al., 1998-a; MUSOLL; LANG; CORTADELLA, 1998) exploram uma redundância espacial, ou seja, eles aumentam a quantidade de linhas de comunicação. Outros esquemas exploram a redundância temporal, ou seja, eles aumentam a quantidade de bits transmitidos em sucessivos ciclos de *clock* (STAN; BURLESON, 1997-b). Mesmo aumentando a quantidade de linhas de comunicação ou de bits transmitidos, estes esquemas conseguem reduzir a quantidade de chaveamentos nas linhas do barramento, reduzindo o consumo de potência nas mesmas. Existem também alguns esquemas que não utilizam redundância espacial nem temporal, e funcionam com base em observações estatísticas do tráfego transmitido (MEHTA; OWENS; IRWIN, 1996; BENINI et al., 1998-b).

Em (COSTA, 2002) o autor investiga alguns desses esquemas de codificação visando reduzir o consumo de potência em operadores aritméticos que são utilizados como módulos básicos nos circuitos de filtros FIR e FFT. Neste caso, os esquemas mais eficientes são os que exploram uma alta correlação entre os dados sucessivos de entrada. A correlação tende a ser maior nos barramentos de endereços, onde geralmente o dado é incrementado a cada transmissão. Já nos barramentos de dados, a correlação é menor do que nos barramentos de endereços.

Alguns esquemas de codificação requerem um conhecimento prévio dos parâmetros estatísticos do tráfego de entrada, o que nem sempre se faz disponível. Este trabalho enfatiza somente os esquemas de codificação que não requerem tal conhecimento, pois o objetivo aqui é aplicá-los a sistemas baseados em *Networks-on-Chip*, independentemente do tipo de tráfego.

As seções 3.1 a 3.6 descrevem esquemas de codificação encontrados na literatura. A seção 3.7 apresenta o *T-Bus-Invert*, o esquema de codificação proposto neste trabalho. A seção 3.8 introduz a abordagem que insere esquemas de codificação em sistemas que utilizam *Networks-on-Chip* como infra-estrutura de comunicação.

3.1 Codificação *Gray*

A codificação *Gray* é um dos métodos mais utilizados para redução de transições de sinais em barramentos de endereços (MEHTA; OWENS; IRWIN, 1996). Uma seqüência de números consecutivos (incrementados), quando codificados no método *Gray*, apresenta em cada palavra somente um bit diferente, com relação à palavra anterior. Assim, é possível reduzir em até 50% o número de transições em relação ao código binário original, fazendo com que o método *Gray* seja bastante eficiente quando os dados são seqüenciais ou apresentam um alto grau de correlação.

A conversão do código binário para o código *Gray* é feita de acordo com as Equações (1) e (2) (CHANDRAKASAN; BRODERSEN, 1995), onde B = (b_{n-1}, b_{n-2},..., b_1, b_0) é a representação binária do número e G = (g_{n-1}, g_{n-2},..., g_1, g_0) é a representação do número em código *Gray*. Esta conversão consiste em repetir o bit mais significativo da palavra binária e utilizar operações lógicas *xor* entre todos os bits consecutivos da palavra.

$$g_{n-1} = b_{n-1} \tag{1}$$

$$g_i = b_{i+1} \; xor \; b_i \quad (i = n-2,\ldots,0) \tag{2}$$

A conversão do código *Gray* em código binário é realizada através das Equações (3) e (4). Esta conversão também é realizada repetindo-se o bit mais significativo da palavra em código *Gray* e utilizando operações *xor*. Entretanto, cada bit a ser convertido depende da conversão anterior, o que cria um aumento do caminho crítico, bem como da complexidade, se comparada com a conversão inversa.

$$bn\text{-}1 = gn\text{-}1 \tag{3}$$

$$bi = bi+1 \text{ xor } gi \quad (i = n\text{-}2,\ldots,0) \tag{4}$$

A Tabela 3.1 mostra um exemplo de funcionamento do método *Gray* em um canal de comunicação com largura de 8 bits. A primeira coluna contém os dados no formato binário original. A segunda coluna apresenta o número de transições no canal de comunicação após cada transmissão do dado original. A terceira coluna contém os dados transmitidos no canal de comunicação, codificados de acordo com o método *Gray*. Por fim, a quarta coluna apresenta o número de transições após cada transmissão do dado codificado. No exemplo é possível observar a existência de somente uma transição quando o dado transmitido é um valor consecutivo ao anterior. Na sexta transmissão do exemplo, o número de transições é igual a dois, pois o dado atual e o anterior não são valores consecutivos.

Uma vantagem deste método é o fato de ele apresentar uma boa eficiência na redução do número de transições sem necessitar de linhas adicionais no canal de comunicação. Em (SU; DESPAIN, 1995) os autores afirmam que o uso do método *Gray* em endereços de memórias *cache* pode reduzir o consumo de potência em até 30% nas *caches* de instrução.

Tabela 3.1: Exemplo de funcionamento do método *Gray* em um canal de comunicação com largura de 8 bits.

Dado original	Número de Transições	Codificação *Gray*	Número de Transições
00000100	-	00000110	-
00000101	1	00000111	1
00000110	2	00000101	1
00000111	1	00000100	1
00001000	4	00001100	1
00000110	3	00000101	2
00000111	1	00000100	1
00001000	4	00001100	1
Total = 16 Transições		**Total** = 8 Transições	

3.2 Codificação *Transition*

Nesta técnica, os dados recebidos pelo codificador são comparados com os dados recebidos anteriormente, os quais ficam armazenados em um registrador. Consideremos B = (b_{n-1}, b_{n-2},..., b_1, b_0) o conjunto de todos os bits de um dado original o qual se deseja transmitir em um barramento de largura n, R = (r_{n-1}, r_{n-2},..., r_1, r_0) o conjunto de todos os bits armazenados no registrador, correspondentes ao dado recebido anteriormente, e T = (t_{n-1}, t_{n-2},..., t_1, t_0) o dado codificado, transmitido no barramento. A técnica *Transition* (RAMOS; OLIVEIRA, 1999) implica em transmitir um sinal em '1' lógico para cada bit i (onde i = ($n-1, n-2,...,0$)) toda vez que houver uma transição do bit i armazenado (r_i) para o bit i do dado atual (b_i). Quando não existe transição é enviado um sinal em '0' lógico em t_i. A codificação *Transition* é realizada através da Equação (5). A decodificação do método *Transition* é realizada de acordo com a Equação (6). Neste caso, o bit decodificado é o resultado da operação *xor* entre o bit transmitido (t_i) e o último bit decodificado (b_i).

$$t_i = r_i \; xor \; b_i \qquad (i = n-2,\ldots,0) \tag{5}$$

$$b_i = r_i \; xor \; t_i \qquad (i = n-2,\ldots,0) \tag{6}$$

A Tabela 3.2 mostra um exemplo de funcionamento do método *Transition* em um canal de comunicação com largura de 8 bits, apresentando os dados originais e codificados, bem como o número de transições obtido a cada transmissão, tanto do dado original quanto do dado codificado. No exemplo da Tabela 3.2, o uso do método *Transition* propiciou uma redução de 21% no número de transições no canal de comunicação. Em (RAMOS; OLIVEIRA, 1999) os autores mostram que este método é eficiente na redução da atividade de transição quando aplicado em barramentos de dados, tanto para a configuração de *cache* unificada quanto dual (dados e instruções). Entretanto, o método não é eficiente quando aplicado a barramentos de endereço, devido ao alto grau de correlação entre os dados.

A vantagem do método *Transition* é a simplicidade, necessitando apenas de um registrador de tamanho n bits (onde n corresponde à largura do canal de comunicação), bem como n portas *xor* tanto no módulo de codificação quanto no módulo de decodificação.

Tabela 3.2: Exemplo de funcionamento do método *Transition* em um canal de comunicação com largura de 8 bits.

Dado original	Número de Transições	Codificação *Transition*	Número de Transições
01101001	-	01101001	-
00110110	6	01011111	4
10010110	2	10100000	8
10101001	6	00111111	6
01011110	7	11110111	3
00100101	6	01111011	3
11011110	7	11111011	1
11101011	4	00110101	5
Total = 38 Transições		**Total** = 30 Transições	

3.3 Codificação *Bit Prediction*

O método de codificação *Bit Prediction* parte do princípio de que alguns barramentos apresentam padrões de transmissão bastante regulares (RAMOS; OLIVEIRA, 1999). Por exemplo, barramentos de endereço apresentam uma alta percentagem de endereços seqüenciais. Este fator pode ser utilizado para fazer a predição do próximo dado no barramento com uma precisão razoável.

A predição do valor de um bit de dados pode ser realizada a partir da utilização de uma tabela com 2^k entradas, indexadas pelos últimos k valores do bit nas palavras anteriores. Quando a predição é correta, não acontece nenhuma transição no barramento. Quando a predição não é correta, a transição no barramento indica que o valor na tabela está errado, que este valor deve ser complementado, e que a tabela deve ser atualizada. A estrutura de hardware necessária para implementação do método *Bit Prediction* é mostrada na Figura 3.1.

As vantagens deste método são o fato de não necessitar de linhas extra de barramento para sua implementação, bem como o fato de não inserir nenhum atraso significativo nos sinais do barramento. Sua desvantagem é a necessidade de um hardware adicional mais complexo (tabelas e registradores) para sua implementação.

Em (RAMOS; OLIVEIRA, 1999) os autores aplicam o método *Bit Prediction* em barramentos de dados e de endereços. O método apresentou valores de redução de atividade de transição de ate 50% quando aplicado a barramentos de endereços, onde existe uma seqüencialidade nos dados. Entretanto, o método não se mostra eficiente quando aplicado a barramentos de dados, não sendo indicado para este tipo de aplicação. Também é importante observar que a estrutura para implementação do método utiliza uma tabela com 2^k entradas para predição de bits. Sendo assim, este método só é praticável para valores pequenos de k.

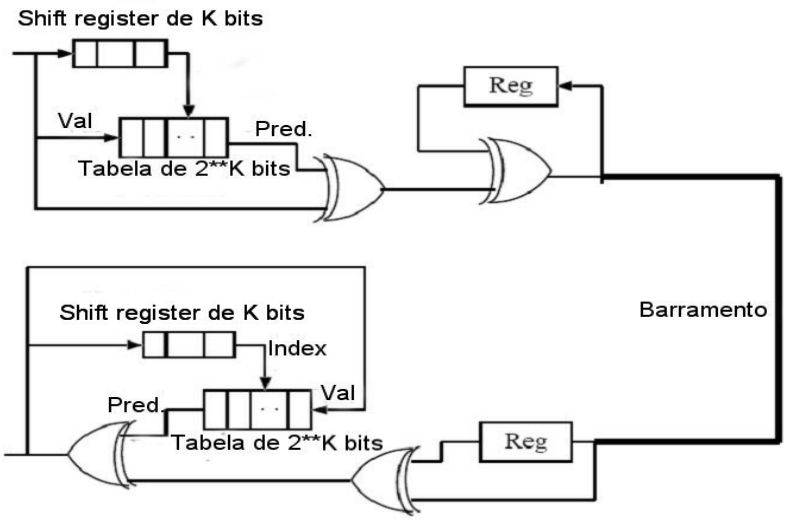


Figura 3.1: Estrutura para implementação do método *Bit Prediction*.

3.4 Codificação *Limited Weight*

A idéia básica utilizada no método *Limited Weight* é reduzir ao máximo possível a ocorrência de bits em ‘1’ lógico nos dados transmitidos no barramento. Para isto, é necessário utilizar barramentos com maior quantidades de linhas de transmissão e escolher códigos que transportam poucos bits em ‘1’ lógico e a grande maioria em ‘0’ lógico. Esta técnica foi proposta por (STAN, M. R.; BURLESON, 1997-a), que utiliza o termo “peso” (*weight*) de uma palavra codificada para representar o número de bits iguais a ‘1’ desta palavra.

Por definição, se o número de bits de uma palavra de dados é *n*, então a aplicação da técnica *m-Limited Weight Coding* (*m-LWC*) pode resultar em palavras codificadas com peso $\leq m$, onde *m* corresponde ao número máximo de bits em ‘1’ lógico em uma palavra codificada. A

Tabela 3.3 mostra um exemplo de aplicação da técnica *Limited Weight* para valores de *m* iguais a 1 e 2 em uma seqüência de palavras de *n* igual a 4 bits.

Na

Tabela 3.3 pode-se observar que a aplicação da técnica 1-LWC resulta em palavras codificadas com no máximo 1 bit em ‘1’ lógico, enquanto que a técnica 2-LWC resulta em palavras codificadas com no máximo 2 bits em ‘1’ lógico. Em ambos os casos, é necessário que se utilize um número maior de linhas de comunicação no barramento, de forma a transmitir as palavras codificadas. Também é possível observar que, quanto menor a quantidade de bits em ‘1’ lógico, maior é quantidade de linhas de comunicação necessárias para codificação do dado.

De acordo com (STAN; BURLESON, 1997-b), a técnica 1-LWC (mínimo possível) dificilmente é utilizada na prática, exceto para valores pequenos de *n*, devido ao crescimento exponencial do número de linhas de comunicação necessárias no barramento.

Tabela 3.3: Exemplo de aplicação do método *Limited Weight*.

Decimal	Binário	2-LWC	1-LWC
0	0000	00000	000000000000000
1	0001	00001	000000000000001
2	0010	00010	000000000000010
3	0011	00011	000000000000100
4	0100	00100	000000000001000
5	0101	00101	000000000010000
6	0110	00110	000000000100000
7	0111	11000	000000001000000
8	1000	01000	000000010000000
9	1001	01001	000000100000000
10	1010	01010	000001000000000
11	1011	10100	000010000000000
12	1100	01100	000100000000000
13	1101	10010	001000000000000
14	1110	10001	010000000000000
15	1111	10000	100000000000000

3.5 Codificação *Bus-Invert*

O método *Bus-Invert*, proposto em (STAN; BURLESON, 1995), é um esquema de codificação que não necessita nenhum conhecimento *a-priori* do tráfego a ser codificado. Este método utiliza uma linha de controle a mais no canal de comunicação, chamada de *invert*. Por convenção, quando *invert* = '0', o valor nas linhas de comunicação corresponde ao dado transmitido na forma original. Quando *invert* = '1', o valor nas linhas de comunicação corresponde ao complemento do dado transmitido.

O método *Bus-Invert* funciona da seguinte forma. Para cada dado a ser transmitido, é calculada a distância de *Hamming* (a quantidade de bits diferentes) entre este dado e o dado atual no canal de comunicação (ou seja, o último dado que foi transmitido). Neste cálculo, deve-se levar em consideração também o valor atual do bit de controle (*invert*). Se a distância de *Hamming* é maior do que $n/2$, onde n é a largura do canal de comunicação, *invert* deve ser igual a '1' e o complemento do dado deve ser transmitido. Caso contrário, *invert* deve ser igual a '0' e o dado transmitido sem alteração. No lado do receptor, o dado recebido deve ser complementado quando *invert* = '1'.

A Tabela 3.4 mostra o funcionamento do método *Bus-Invert* em um canal de comunicação com largura de 8 bits. A primeira coluna contém o dado a ser transmitido no canal. A segunda coluna mostra a distância de *Hamming* entre o dado a ser transmitido e o último dado transmitido. A terceira coluna contém o dado a ser transmitido no barramento (tendo como bit mais significativo o bit extra de controle, *inv*) e a quarta coluna mostra a quantidade de transições com relação ao dado transmitido anteriormente.

No exemplo, o primeiro dado a ser enviado é “00110100”. Sendo este o primeiro dado, a distância de *Hamming* não é calculada e o dado original é enviado no barramento. O segundo dado a ser transmitido é “00101000”, com distância de *Hamming* igual a 3 com relação ao dado anterior. Neste caso, o dado é novamente transmitido sem inversão, pois a distância de *Hamming* não é maior que 4. Assim, acontecem 3 transições nesta transmissão, pois 3 fios mudaram seu estado lógico com relação ao dado anterior. O terceiro dado a ser transmitido é “10010011”, apresentando distância de *Hamming* igual a 6. Neste caso o dado é invertido antes de ser transmitido (“01101100”) e o bit de controle é setado em ‘1’. Enviando o dado invertido, o número de transições na transmissão diminui de 6 para 3, mesmo contando com o bit de controle, que insere uma transição a mais. O quarto dado a ser transmitido é “10010000”, com distância de *Hamming* igual a 7. É importante observar que a distância de *Hamming* é calculada entre o dado a transmitir (“0 10010000”) e o último dado no barramento (“1 01101100”). Este dado é então invertido e transmitido, resultando em 2 transições, ao invés de 7 (com o dado original).

Tabela 3.4: Exemplo de funcionamento do método *Bus-Invert* em um canal de comunicação com largura de 8 bits.

Dado original	Distância de *Hamming*	Codificação *Bus/Invert* (inv)	Número de Transições
00110100	-	0 00110100	-
00101000	3	0 00101000	3
10010011	6	1 01101100	3
10010000	7	1 01101111	2

Este método faz com que o número máximo de transições possíveis seja reduzido de *n* para *n*/2, reduzindo também a dissipação de potência de pico pela metade. Por outro lado, o número médio de transições não chega a ser reduzido pela metade. Primeiro por que a linha *invert* insere novas transições na transmissão. Segundo, por que a distância de *Hamming* para o próximo dado não é uniforme. Ou seja, a probabilidade de que o próximo valor seja diferente do atual em 1, 2, 3, 4, 5, 6, 7 ou 8 bits (para uma largura de *flit* igual a 8 bits) é uma distribuição binomial onde a maior probabilidade é para uma distância de *Hamming* igual a 4 (STAN; BURLESON, 1995). Entretanto, 4 (ou *n*/2) é o valor para o qual não existe ganho algum ao inverter-se o dado a ser transmitido.

A Figura 3.2 ilustra o esquema básico do codificador, composto por dois blocos. O primeiro implementa uma função de controle, indicando quando o dado a ser transmitido deve ou não ser invertido, enquanto que o segundo é responsável por inverter o dado de acordo com o sinal de controle. O bloco de controle consiste de duas partes: (*i*) um conjunto de *n* portas *xor* que recebem como entradas o estado lógico atual e o próximo a transmitir, para cada bit do canal de comunicação (a porta terá ‘1’ na saída no caso de alteração do valor) e (*ii*) uma função de controle que indica se a maioria dos bits alterou seu valor com relação à palavra anterior, levando em consideração o bit de controle da palavra anterior. Esta função tem como saída o sinal *invert*, que além de ser o sinal de controle do bloco inversor, é o próprio bit de controle transmitido através do canal e dados. A Figura 3.3 ilustra uma possível implementação para a função de controle, utilizando uma árvore de somadores (*full-adders*). A mesma

tem como entradas as *n* portas *xor* e o bit de controle da palavra anterior, transmitido no tempo *t* - 1. No exemplo do inversor que opera sobre palavras com largura de 8 bits, a função de controle indica se houve mudança em 5 ou mais bits, ativando a função inversora.

O decodificador é composto somente pelo bloco inversor, tendo como entrada de controle o próprio sinal *inverter* transmitido pelo canal de comunicação.

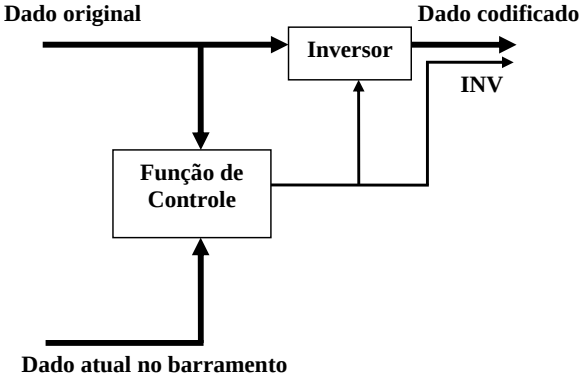


Figura 3.2: Esquema básico do codificador *Bus-Invert*.

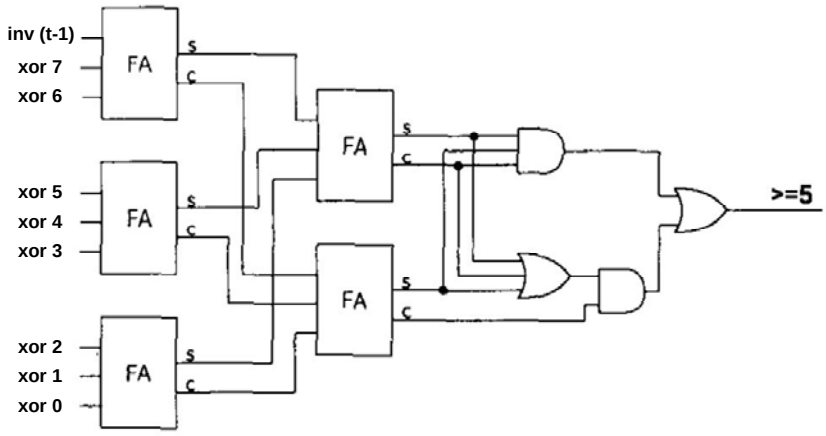


Figura 3.3: Possível implementação da função de controle (STAN; BURLESON, 1995).

Quando se utiliza canais de comunicação com larguras maiores, como por exemplo 16 ou 32 bits, pode ser vantajoso agrupar as linhas de comunicação em *clusters*, inserindo uma linha a mais de controle para cada *cluster*. A Figura 3.4 ilustra um barramento de 16 bits dividido em 2 *clusters* de 8 bits, com 1 bit de controle para cada *cluster*. Neste exemplo os bits de controle foram inseridos após o bit mais significativo do barramento. Esta é uma forma de facilitar a alteração do projeto do barramento original, porém, nada impede que cada bit de controle seja inserido após o bit mais significativo de seu respectivo *cluster*. Em (STAN; BURLESON, 1995), os autores afirmam que, em barramentos, a solução que apresenta a melhor relação custo/benefício é a que utiliza agrupamentos de 8 bits.

A Tabela 3.5 mostra um exemplo de funcionamento do método *Bus-Invert* em um canal de comunicação com largura de 16 bits, com 1 bit de controle. Neste exemplo, o

primeiro dado transmitido no barramento é “0011010010100100”. O segundo dado a ser transmitido é “0011100100001011” distância de *Hamming* igual 9. Como a distância de *Hamming* é maior do que *n*/2, o dado é invertido antes da transmissão. Esta inversão faz com que a quantidade de transições na transmissão seja 8, em vez de 9 com o dado original.

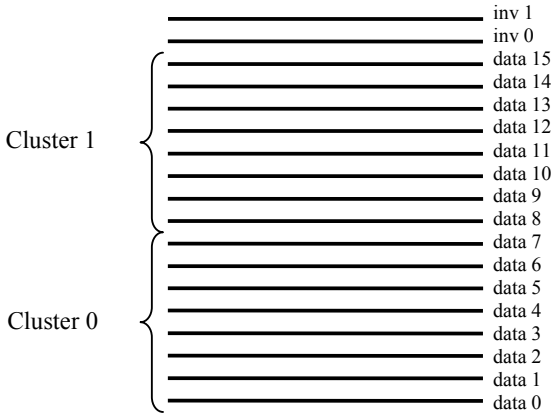


Figura 3.4: Exemplo de barramento de 16 bits dividido em 2 *clusters* de 8 bits.

Analisemos, então, um exemplo com os mesmos dados de 16 bits do exemplo anterior, mas agora em um barramento dividido em 2 *clusters*, como mostra a

Tabela 3.6. Neste caso, as distâncias de *Hamming* devem ser calculadas independentemente para cada *cluster*. O primeiro dado é enviado no seu formato original, porém dividido em 2 partes, a metade mais significativa no *cluster1* e a metade menos significativa no *cluster0*. Na segunda transmissão, o dado a ser enviado é “0011100100001011”, separado em “00111001” e “00001011”, com distâncias de *Hamming* igual a 3 e 6. Note que a soma das duas distâncias de *Hamming* é igual à distância de *Hamming* do exemplo sem divisão em *clusters* (9), pois se trata das mesmas palavras de 16 bits sendo transmitidas. Observando que, agora estão sendo utilizados *clusters* de 8 bits, a inversão deve ser feita quando a distâncias de *Hamming* é maior que 4. Assim sendo, somente o dado no *cluster0* deve ser invertido, enquanto que o dado no *cluster1* é enviado no seu formato original. Neste caso, o número de transições é reduzido de 9 para 6, em vez de 9 para 8 obtidos no exemplo anterior, mostrando uma eficiência maior no barramento dividido em clusters. Entretanto, um fato importante que deve ser analisado é o aumento de área e potência devido à inserção de *m* fios a mais no barramento, onde *m* corresponde ao número de *clusters* utilizados.

Tabela 3.5: Exemplo de funcionamento do método *Bus-Invert* em um canal de comunicação com largura de 16 bits, com 1 bit de controle.

Dado original	Distância de *Hamming*	Codificação *Bus/Invert* (inv)	Número de Transições
0011010010100100	-	0 0011010010100100	-
0011100100001011	9	1 1100011011110100	8

Tabela 3.6: Exemplo de funcionamento do método *Bus-Invert* em um canal de comunicação com largura de 16 bits dividido em dois *clusters*, com 2 bits de controle.

Dado original (Cluster1) (Cluster0)	Distância de *Hamming*	Codificação *Bus/Invert* (inv1)(inv0)	Número de Transições
00110100 10100100	- -	0 0 00110100 10100100	-
00111001 00001011	3 6	0 1 00111001 11110100	6

3.6 Codificação *Adaptive Probability Encoding*

Em (BENINI et al, 2000), é proposta uma arquitetura geral de codificação/decodificação que reduz a quantidade de transições nos sinais de comunicação e também usa um par *decorrelator*/*correlator* para reduzir a quantidade de bits '1' transmitidos.

A Figura 3.5 ilustra esta arquitetura geral de codificação/decodificação. O codificador (E) recebe uma seqüência de palavras $x(n)$ (n = 0, 1, 2,...) com W-bits de largura. Ele consiste de três blocos: *(i)* um registrador que armazena $x(n\text{-}1)$ quando a entrada do codificador é $x(n)$; *(ii)* uma função combinacional de codificação que gera a palavra codificada $y(n)$ em função de $x(n)$ e $x(n\text{-}1)$; *(iii)* um *decorrelator* (STAN; BURLESON, 1997-b) que transforma transições de $y(n)$ em bits de valor 1 nas linhas de comunicação, gerando $z(n)$ (bits de valor 0 correspondem a valores estacionários nas linhas de comunicação).

O decodificador (D) recebe como entrada uma palavra $z(n)$ transmitida através do canal de comunicação e computa a palavra $x(n)$ original. Ele consiste de três blocos: *(i)* um *correlator* que executa a função inversa do *decorrelator*, reproduzindo $y(n)$; *(ii)* uma função combinacional de decodificação que reproduz $x(n)$ a partir de $y(n)$ e $x(n\text{-}1)$; *(iii)* um registrador que armazena $x(n\text{-}1)$ quando a saída do decodificador é $x(n)$.

O *decorrelator* executa a função $out(n) = in(n)$ *xor* $out(n\text{-}1)$, enquanto que o *correlator* executa a função $out(n) = in(n)$ *xor* $in(n\text{-}1)$. A vantagem de se usar um par *decorrelator*/*correlator* é que ele transforma o problema de minimizar o número de transições nas linhas de comunicação no problema de minimizar a quantidade de bits '1' na entrada do *decorrelator* (STAN; BURLESON, 1997-b).

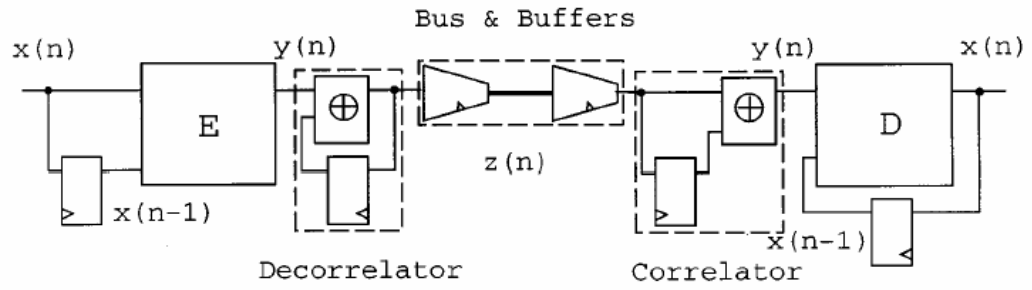


Figura 3.5: Arquitetura geral de codificação/decodificação.

A função de codificação *E* tem como objetivo minimizar a quantidade média de bits '1' na sua entrada, e ao mesmo tempo garantir que a palavra codificada $y(n)$ seja decodificada de forma não ambígua pela função *D*. A arquitetura apresentada na Figura

3.5 pode ter diferentes configurações, de acordo com a definição das funções de codificação e decodificação.

Em (BENINI et al, 2000), os autores apresentam um algoritmo utilizado para preencher uma tabela de codificação que é utilizada pelo codificador para gerar a palavra $y(n)$, e pelo decodificador para reproduzir a palavra original $x(n)$. A idéia nesta abordagem é enviar uma palavra codificada $y(n)$ com a quantidade mínima possível de bits '1', de acordo com as palavras de entrada $x(n)$ e $x(n\text{-}1)$. Para isto, é necessário que se conheça a distribuição probabilística de cada par possível de palavras consecutivas. Os autores chamam esta distribuição de *Joint Probability Distribution* (JPD). A tabela é então organizada em ordem decrescente de probabilidade, de forma que os pares com maior probabilidade ficam no topo, assim como as saídas com menor quantidade de bits '1'.

Entretanto, esta abordagem tem algumas limitações. Uma delas é o fato de que o conhecimento do JPD pode ser incompleto ou aproximado, pois obter uma estimativa $P_{xi,xj}$ precisa para cada par de palavras de entrada consecutivas pode tornar-se impraticável para grandes fluxos de dados. Além disso, a implementação do codificador/decodificador pode ser inaceitável quanto à sua área e consumo de potência, devido à complexidade da tabela de codificação com 3 colunas ($x(n)$, $x(n\text{-}1)$ e $y(n)$) e 2^{2W} linhas.

No mesmo trabalho, os autores propõem um esquema de codificação probabilístico adaptativo (*Adaptive Probability Encoding*), o qual não requer nenhum conhecimento *a-priori* do tráfego, sendo capaz de adaptar-se em tempo de execução. Este esquema opera bit-a-bit, em vez de palavra-a-palavra, ignorando a correlação espacial entre bits de uma mesma palavra.

A codificação é feita com base em informações estatísticas aproximadas coletadas através da observação da seqüência de bits em uma janela de tamanho fixo S. Os autores afirmam que uma janela de tamanho $S = 64$ oferece o melhor compromisso entre complexidade e precisão.

Estas informações estatísticas dizem respeito às probabilidades de ocorrência dos quatro possíveis pares de valores consecutivos de um único bit ($P_{0,0}$, $P_{0,1}$, $P_{1,0}$ e $P_{1,1}$). A fim de lidar com valores inteiros e conseqüentemente simplificar o hardware, os autores usam a freqüência das ocorrências N_{00}, N_{01}, N_{10} e N_{11} em vez das probabilidades. Sendo que, as somas de todas as freqüências é conhecida ($N_{00} + N_{01} + N_{10} + N_{11} = S - 1$), nem todas as quatro são realmente necessárias para o funcionamento do sistema. Considerando-se que N_{01} e N_{10} devem ser balanceadas dentro da janela de observação, podendo diferir uma da outra de no máximo '1', é suficiente considerar somente as freqüências N_{00} e N_{11}, já que seu conhecimento infere os outros dois. Além disso, sendo $N_{01} = N_{10}$, os autores utilizam o símbolo N_T para designá-las.

Este esquema utiliza quatro funções diferentes de codificação $F(x(n), x(n\text{-}1))$ para determinar y(n). A seleção da função a ser utilizada a cada instante é feita de acordo com as freqüências que estimam as probabilidades da janela de observação atual. A decisão é tomada da seguinte forma:

a) $y(n) = x(n)$quando $N_{00} > N_T > N_{11}$

b) $y(n) = x(n)$'quando $N_{11} > N_T > N_{00}$

c) $y(n) = x(n)$ *xor* $x(n\text{-}1)$quando $N_T < \{ N_{11}, N_{00}\}$

d) $y(n) = x(n)$ *nxor* $x(n\text{-}1)$quando $N_T > \{ N_{11}, N_{00}\}$

Quando o par de símbolos mais provável é “00”, a melhor decisão é manter o valor atual do bit, enviando ‘0’s na linha de comunicação. Da mesma forma, quando o par mais provável é “11”, o codificador inverte o dado de entrada, enviando ‘0’s. Por outro lado, quando o par mais provável é N_T, as transições são eliminadas através de uma função *xor* entre o bit atual e o anterior. Isto gera uma seqüência de bits ‘1’, que são complementados antes de serem enviados na linha de comunicação. Vale lembrar que este codificador tem como objetivo não somente diminuir a quantidade de transições no fio, mas também a quantidade de bits ‘1’ transmitidos.

A Figura 3.6 ilustra a arquitetura do codificador adaptativo de 1 bit. A entrada $x(n)$ e seu valor anterior $x(n\text{-}1)$ alimentam os dois contadores que armazenam a quantidade de ocorrências N_{00} e N_{11}. Um contador chamado *WinCnt* controla o início e fim de cada janela de observação e reinicializa os contadores N_{00} e N_{11} a cada S ciclos. O contador *WinCnt* é compartilhado por todos os codificadores do canal de comunicação, de forma que este seja utilizado para reinicializar ao mesmo tempo todos os contadores de todos os codificadores de 1 bit. A área sombreada à direita na Figura 3.6 computa a função de codificação, com base no conhecimento de $x(n)$, $x(n\text{-}1)$, N_{00} e N_{11}.

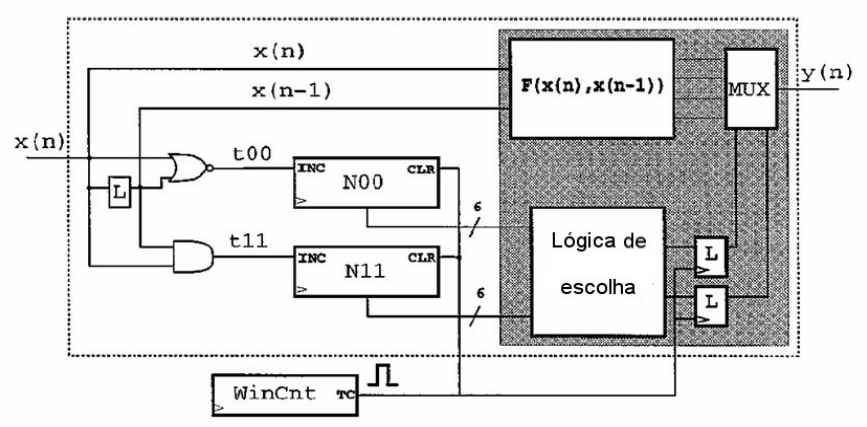


Figura 3.6: Arquitetura do codificador adaptativo.

A arquitetura do decodificador, mostrada na Figura 3.7 é similar à do codificador, sendo que as ocorrências de N_{00} e N_{11} são computadas com base na observação dos pares de valores consecutivos de saída do decodificador ($x(n)$, $x(n\text{-}1)$). Além disso, as funções de decodificação devem receber como entrada $y(n)$ e $x(n\text{-}1)$, executando a função inversa à do codificador. Por exemplo, se a função de codificação for $y(n) = x(n)$ *xor* $x(n\text{-}1)$, a função de decodificação deve ser $x(n) = y(n)$ *xor* $x(n\text{-}1)$.

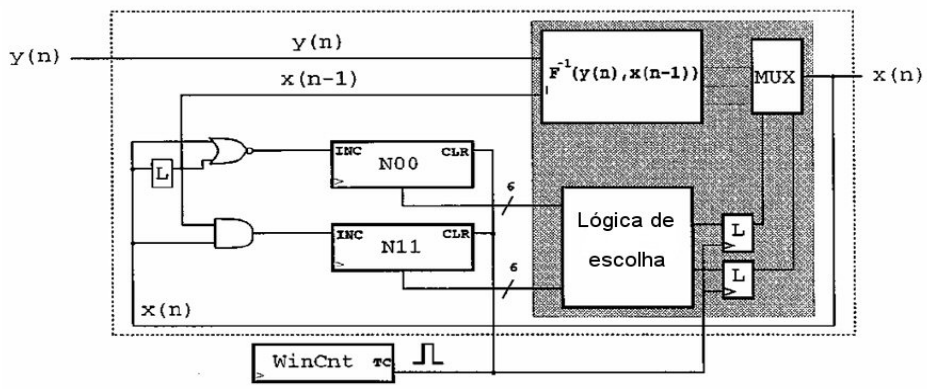


Figura 3.7: Arquitetura do decodificador adaptativo.

3.7 Codificação *T-Bus-Invert*

O *T-Bus-Invert* é uma das contribuições deste trabalho. Este esquema de codificação utiliza os princípios básicos do esquema *Bus-Invert*, ou seja, ele inverte o dado a ser transmitido quando a distância de *Hamming* entre este dado e o último dado transmitido é maior do que metade do número de bits deste dado. Entretanto, o *T-Bus-Invert* não requer a inserção de linhas extras na estrutura de comunicação, como acontece no *Bus-Invert*. Esta inserção acabaria refletindo em todos os módulos da NoC, requerendo a alteração dos mesmos e, consequentemente, aumentando o consumo de potência em todos estes módulos. O consumo de potência nas diferentes configurações da NoC original e na NoC modificada para o *Bus-Invert*, bem como o consumo de potência nos módulos dos diferentes esquemas de codificação serão abordados na seção 4.4.

O esquema *T-Bus-Invert* utiliza o bit mais significativo do canal de dados como bit de controle (*invert*). Consequentemente, em um canal de comunicação com *flits* de *n* bits, *n*-1 bits de dados são enviados em cada *flit*. A Tabela 3.7 ilustra um exemplo de funcionamento do *T-Bus-Invert* em uma estrutura de comunicação com largura de *flit* igual a 8 bits. A primeira coluna da tabela corresponde ao estado atual da máquina-de-estados que controla a codificação. A segunda coluna apresenta os dados originais, recebidos pelo codificador. A terceira coluna mostra o conteúdo do *buffer* temporário, onde são armazenados, durante 1 ciclo de clock, os bits em espera que serão codificados e transmitidos no próximo ciclo de clock. A quarta coluna mostra o dado a ser codificado, o qual pode ser composto por bits do dado original recebido naquele ciclo de clock, por bits do *buffer* temporário ou por uma combinação dos mesmos. Na quinta coluna é calculada a distância de *Hamming* entre o dado a codificar e o último dado codificado. Por fim, a sexta coluna mostra o dado codificado e a sétima coluna mostra a quantidade de transições obtidas após a codificação.

No primeiro flit, quando a máquina-de-estados está no estado ‘0’, a codificação é feita sobre os 7 bits menos significativos do dado original, enquando que o bit mais significativo desta palavra é armazenado no *buffer* temporário, para ser codificado e transmitido no próximo ciclo de clock, como mostra a Figura 3.8. No estado ‘1’, ilustrado na Figura 3.9, a codificação é feita sobre a palavra formada pelo bit armazenado no *buffer*, concatenado com os 6 bits menos significativos do novo dado original recebido, enquanto que os 2 bits mais significativos do dado original são armazenados no *buffer* temporário. No estado ‘2’ o dado a codificar é formado pelos 2

bits que se encontram no *buffer*, concatenados com os 5 bits menos significativos do dado original. Os 3 bits mais significativos do dado original são armazenados no *buffer*. Este processo continua até o estado '6', onde os 7 bits mais significativos do dado original são armazenados no *buffer*. No estado '7' o módulo de codificação do *T-Bus-Invert* envia um sinal parando o recebimento de dados durante 1 ciclo de clock para, então, executar a codificação sobre os 7 bits armazenados no *buffer* e transmiti-los como um novo dado. No próximo ciclo de clock a máquina-de-estados volta para o estado '0' e o processo se repete.

Tabela 3.7: Exemplo de funcionamento do esquema *T-Bus-Invert* em uma estrutura de comunicação com largura de *flit* igual a 8 bits.

Estado	Dado Original i7..i0	Buffer b6..b0	Dado a Codificar	Distância de *Hamming*	Dado Codificado (inv)	Número de Transições
0	01001010	i7 0xxxxxx	i6..i0 1001010	-	0 1001010	-
1	10001110	i7..i6 10xxxxx	b6+I5..i0 0001110	2	0 0001110	2
2	01011010	i7..i5 010xxxx	b6..b5+I4..i0 1011010	3	0 1011010	3
3	00000001	i7..i4 0000xxx	b6..b4+I3..i0 0100001	6	1 1011110	2
4	11100011	i7..i3 11100xx	b6..b3+I2..i0 0000011	6	1 1111100	2
5	01010111	i7..i2 010101	b6..b2+I1..i0 1110011	5	1 0001100	3
6	01010100	i7..i1 0101010	b6..b1+ i0 0101010	4	0 0101010	4
7	-	-	b6..b0 0101010	0	0 0101010	0
0	10011110	i7 1xxxxxx	i6..i0 0011110	3	0 0011110	3

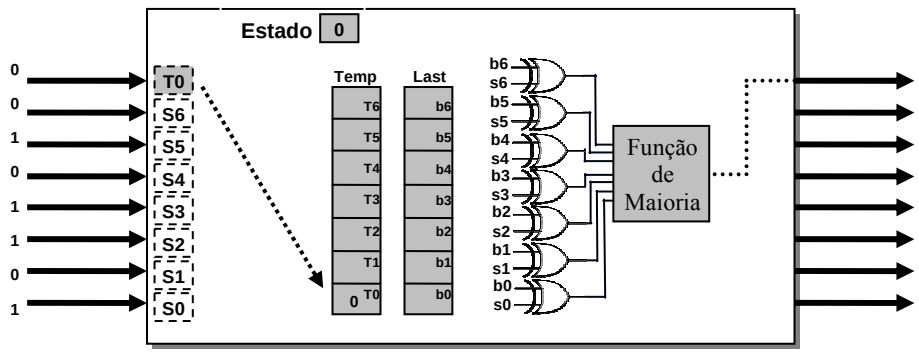


Figura 3.8: Estado 0 da máquina de controle do codificador T-Bus-Invert, onde a codificação é feita sobre os 7 bits menos significativos do dado original, enquando que o bit mais significativo desta palavra é armazenado no *buffer* temporário.

O *T-Bus-Invert* transforma a redundância espacial do esquema *Bus-Invert* em redundância temporal (*Temporal Bus-Invert*). Uma possível desvantagem deste esquema de codificação é uma redução no throughput máximo no lado do *IP Core* que envia os dados para o codificador, devido à latência de 1 ciclo de clock após cada grupo de 7 *flits* transmitidos. Entretanto, esta latência não chega a prejudicar o desempenho das aplicações-alvo, pois as taxas de transmissão dos *IP Cores* são inferiores à taxa de transmissão da NoC. Por exemplo, em uma NoC com largura de *flit* igual a 16 bits operando a 200 MHz, a taxa de transmissão real disponível por canal é de 2,99 Gbps. Esta taxa é calculada considerando-se o tempo necessário para que o roteador execute o mecanismo de arbitragem e o algoritmo de roteamento de cada pacote. Utilizando o esquema *T-Bus-Invert*, esta taxa de transmissão cai para 2,74 Gbps, apresentando uma redução de 8,3%. Em contraste, em uma aplicação que requer uma grande largura de banda, como HDTV (MPEG2), a taxa de transmissão é de 15 Mbps, bem menor do que a taxa de transmissão da NoC.

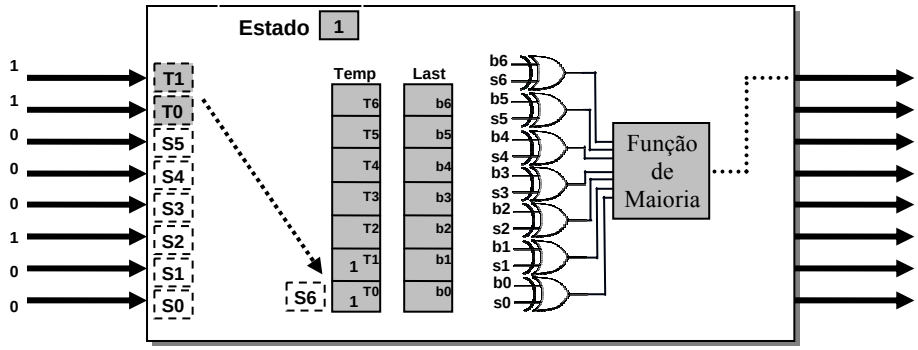


Figura 3.9: Estado 1 da máquina de controle do codificador T-Bus-Invert, onde a codificação é feita sobre a palavra formada pelo bit armazenado no *buffer*, concatenado com os 6 bits menos significativos do novo dado original recebido, enquanto que os 2 bits mais significativos do dado original são armazenados no *buffer* temporário.

3.8 Inserindo esquemas de codificação de dados em *Networks-on-chip*

Em *Networks-on-Chip*, os dados são transmitidos em pacotes, os quais são enviados através de roteadores, partindo de um núcleo de origem, até atingir um núcleo de destino. Estes pacotes são compostos por um cabeçalho (contendo informações de roteamento) e por um corpo (contendo os dados a serem transmitidos). No caso da rede Hermes (MORAES et al., 2004), utilizada como estudo de caso neste trabalho, o cabeçalho é composto por dois *flits*, contendo o endereço de destino do pacote, bem como o tamanho do mesmo. Assim, em uma abordagem que combina esquemas de codificação e uma *Network-on-Chip*, não é vantajoso codificar o cabeçalho do pacote, já que a informação contida no mesmo deve ser lida pela lógica de roteamento de cada roteador, a cada *hop*2 tomado pelo pacote, desde sua origem até seu destino.

2 Um *hop* é a distância entre dois roteadores vizinhos em uma NoC.

Desta forma, as operações de codificação e decodificação podem ser executadas somente em sua origem, convertendo os dados originais em dados codificados, que são transmitidos através da NoC, e no seu destino, recuperando os dados originais.

No caso dos esquemas *Gray, Transition, Adaptive Probability Encoding* e *T-Bus-Invert*, implementados no presente trabalho, os módulos de codificação e decodificação foram inseridos dentro de *wrappers*, que contém também um *IP Core,* e fazem a comunicação deste com as portas locais dos roteadores (ver Figura 3.10). Ou seja, os *wrappers* conectam os *IP Cores* à estrutura de interconexão (NoC). Uma vantagem desta abordagem é que ela pode ser utilizada em qualquer NoC sem alteração da mesma.

O esquema *Bus-Invert* requer alterações na NoC, de forma a inserir um bit de controle em todos os módulos internos e nos canais de comunicação. Quanto ao local para inserção dos módulos de codificação e decodificação, existem duas possibilidades. A primeira opção é inseri-los da mesma forma que para os outros esquemas de codificação implementados neste trabalho, junto às portas locais dos roteadores, utilizando um controle para codificar somente o *payload* dos pacotes e deixando o cabeçalho sem codificação. Outra opção é não utilizar este tipo de controle, codificando todos os *flits* do pacote, inclusive o cabeçalho. Neste caso, é necessário que se utilize uma lógica de inversão em cada roteador, de forma a decodificar somente o cabeçalho do pacote, no caso deste estar invertido, extraindo assim as informações necessárias para o roteamento do pacote. Mesmo utilizando esta segunda opção, os dados do *payload* só precisam ser decodificados ao atingir o seu núcleo de destino. Para o esquema *Bus-Invert*, optou-se por utilizar a segunda opção. Aproveitando-se do fato de que este esquema requer uma modificação em todos os módulos da NoC, inseriu-se também a lógica de inversão em cada roteador. Desta forma, todo o pacote é codificado e evita-se a necessidade de uma lógica de controle a mais no módulo codificador para codificar somente o *payload* e evitar que o cabeçalho seja codificado.

No caso do esquema *T-Bus-invert*, qualquer uma das duas opções também poderiam ser implementadas. Porém, a segunda opção, utilizando decodificadores em cada roteador, não é tão simples como para o *Bus-Invert*. No caso do *T-Bus-invert*, uma lógica de inversão não é suficiente. Seria necessário implementar, em cada roteador, a máquina-de-estados descrita na Tabela 3.7, de forma a decodificar corretamente os *flits* do cabeçalho. Além disso, considerando que este esquema não requer alteração nos módulos da NoC, escolheu-se pela primeira opção, inserindo os módulos dentro dos *wrappers* e utilizando um controle para não codificar o cabeçalho. Esta opção torna o esquema *T-Bus-Invert* independente da NoC, como os outros esquemas previamente citados.

A Figura 3.10 ilustra um *wrapper* conectado à porta local de um roteador Hermes. Os pacotes endereçados ao núcleo local, passam antes pelo módulo *decoder*, antes de serem entregues ao núcleo (PALMA et al., 2006-a). Da mesma forma, os pacotes gerados pelo núcleo local são codificados no módulo *encoder* antes de serem transmitidos pela NoC. As portas restantes do roteador devem ser conectadas a outros roteadores da NoC (não mostrados na figura), de acordo com sua topologia.

Os códigos VHDL com as descrições dos módulos de codificação e decodificação implementados neste trabalho encontram-se nos anexos ao final deste volume.

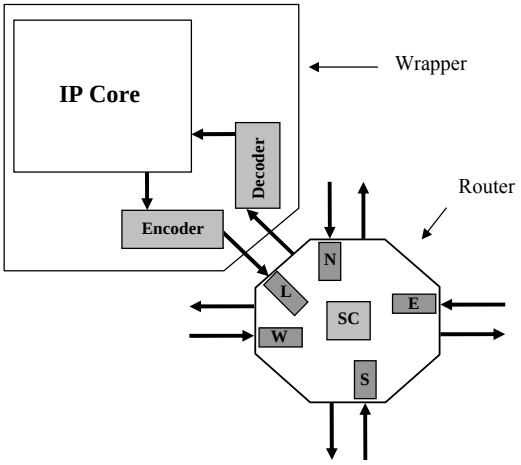


Figura 3.10: Localização dos módulos *encoder* e *decoder*.

3.9 Conclusão

Este capítulo apresentou diferentes esquemas de codificação de dados encontrados na literatura. Dentre os esquemas apresentados, os esquemas *Bit-Prediction* e *Limited Weight* não foram implementados. O primeiro porque utiliza em seus módulos de codificação e decodificação uma tabela de tamanho considerável, o que consumiria área e potência significativas. O segundo porque necessita de muitas linhas de dados, o que aumentaria bastante a largura das palavras nos roteadores, aumentando também seus buffers internos e tornando proibitivos os seus consumos de área e de potência. Os demais esquemas foram implementados e utilizados como estudos de caso neste trabalho, sendo aplicados no contexto de um sistema baseado em NoC.

Neste capítulo também foi apresentada uma das contribuições do trabalho, o esquema *T-Bus-Invert*, um esquema que tira proveito da técnica eficiente do método *Bus-Invert* sem inserir novas linhas no canal de comunicação, ou seja, sem necessitar que a NoC seja alterada. O capitulo também propõe uma abordagem que integra módulos de codificação e de decodificação a sistemas baseados em NoCs.

O capítulo a seguir apresenta o modelo de consumo de potência em NoCs proposto no escopo deste trabalho, o qual foi utilizado para avaliar o consumo dos módulos da NoC e dos módulos dos esquemas de codificação.

4 MODELO DE CONSUMO DE POTÊNCIA EM NOCS

A avaliação do consumo de potência de um circuito determina quanta energia é consumida por operação e quanto calor é dissipado por este circuito. Estes fatores têm uma grande influência em decisões críticas de projeto, tais como: capacidade da fonte de alimentação, tempo de vida da bateria, tamanho das linhas de alimentação, bem como requisitos de encapsulamento e dissipação de calor. Conseqüentemente, a dissipação de potência é uma característica importante em um projeto, pois afeta os custos, a viabilidade e a confiabilidade do mesmo (CHANDRAKASAN; BRODERSEN, 1995).

Na área da computação de alto desempenho, o consumo de potência determina a quantidade de circuitos que podem ser integrados em uma mesma pastilha de silício, bem como a sua freqüência máxima de operação. Além disso, com a crescente popularidade dos sistemas portáteis e da computação distribuída, existem limitações estritas quanto ao consumo médio de potência, pois este é diretamente proporcional ao peso e tamanho da bateria necessária para que o circuito opere por uma certa quantidade de tempo. O consumo médio de potência também é importante quando o problema de projeto em questão é dissipação de calor. Já quando o problema é a definição do tamanho das linhas de alimentação, outra medida de dissipação deve ser considerada: a potência de pico (RABAEY, 1996).

Este capítulo tem por objetivo descrever a forma como foram definidos e criados os modelos de consumo de potência utilizados neste trabalho. Na seção 4.1 é detalhado o fluxo utilizado na aquisição dos parâmetros de consumo de potência para os módulos da NoC e para os módulos dos esquemas de codificação. A seção 4.2 mostra os resultados de área, destes diferentes módulos, com diferentes configurações. A seção 4.3 apresenta a definição do modelo de consumo de potência. A seção 4.4 apresenta os gráficos resultantes da análise sobre o consumo de potência dos vários módulos implementados e simulados no escopo deste trabalho, de acordo com a quantidade de transições de bit em seus sinais de entrada. A seção 4.5 introduz os macromodelos de potência, criados com base nos resultados obtidos com a simulação SPICE.

Por razões de simplicidade, os macromodelos estimam a potência média consumida em um determinado módulo sem considerar seu estado interno, ou seja, a potência é calculada em função da atividade de transições nos sinais de entrada do módulo. Esta simplificação pode implicar em um erro máximo relativamente pequeno, de aproximadamente 5%, segundo experimentos conduzidos neste trabalho. Por exemplo, no caso de uma NoC com largura de *flit* igual a 8 bits e com buffers de 16 palavras, esta

simplificação pode implicar em um erro de no máximo ±1,49 mW na estimativa de consumo de potência no buffer e ±0,25 mW na lógica de controle.

4.1 Aquisição dos Parâmetros de Consumo de Potência

O consumo de potência em um SoC origina-se da operação dos núcleos e dos componentes de interconexão entre estes núcleos. O consumo de potência é proporcional à atividade de chaveamento originada pelos pacotes que se movem através da NoC. Canais de comunicação e roteadores dissipam potência. Vários autores (HU; MARCULESCU, 2003; MURALI; DE MICHELI, 2004; MARCON et al., 2005-a; YE; BENINI; DE MICHELI, 2002; MARCON et al., 2005-b; YE; BENINI; DE MICHELI, 2004) propõem estimar o consumo de potência na NoC através da avaliação do efeito do tráfego em cada componente da mesma.

No caso da infra-estrutura de comunicação da rede Hermes, o elemento básico é um roteador com cinco canais bi-direcionais que o conectam a quatro outros roteadores e a um núcleo local. O roteador Hermes é composto por buffers nas portas de entrada e de uma lógica de controle centralizada, responsável pela arbitragem e roteamento de pacotes. No estudo de caso utilizado neste trabalho, os roteadores são configurados para aplicar o algoritmo de roteamento XY. O consumo de potência do roteador é estimado separando-se o consumo de potência no buffer do consumo de potência na lógica de controle. Também se deve estimar o consumo de potência nos canais de comunicação entre roteadores e entre roteador e núcleo local. Neste trabalho foram considerados comprimentos de 5 mm para os canais entre roteadores e 0.25 mm para os canais locais. Estas dimensões foram escolhidas após ter sido feita a síntese de um roteador Hermes e de um processador Plasma (um processador RISC de 32 bits) (OPENCORES.ORG, 2007), em *standard-cell*. O roteador utilizou uma área de 1 mm^2, enquanto que o processador utilizou uma área de 10 mm^2. Optou-se então por uma dimensão de *tile* um pouco maior, com área de 5 mm x 5 mm.

Nos experimentos conduzidos neste trabalho foi utilizada uma rede Hermes com topologia *mesh*. Os parâmetros de consumo de potência nos componentes da NoC foram obtidos através da variação da largura de *flit* (8, 16 e 32 bits) e da variação da profundidade dos buffers (4, 8 e 16 *flits*) da rede Hermes. Para cada configuração foram simulados envios de diferentes pacotes de 128 *flits* através da NoC, com diferentes padrões de transições de sinais em sua estrutura, variando de 0 a 127 transições em cada linha de comunicação.

Considerando uma abordagem com codificação de dados, é também necessário estimar o consumo de potência nos módulos de codificação e decodificação. Estes módulos também foram simulados com os mesmos padrões de tráfego utilizados na simulação da NoC, e com larguras de *flit* iguais a 8, 16 e 32 bits.

As estimativas de consumo de potência são realizadas no nível elétrico. O fluxo utilizado para a aquisição dos parâmetros de consumo de potência dos módulos, ilustrado na Figura 4.1, consiste de três etapas, descritas a seguir.

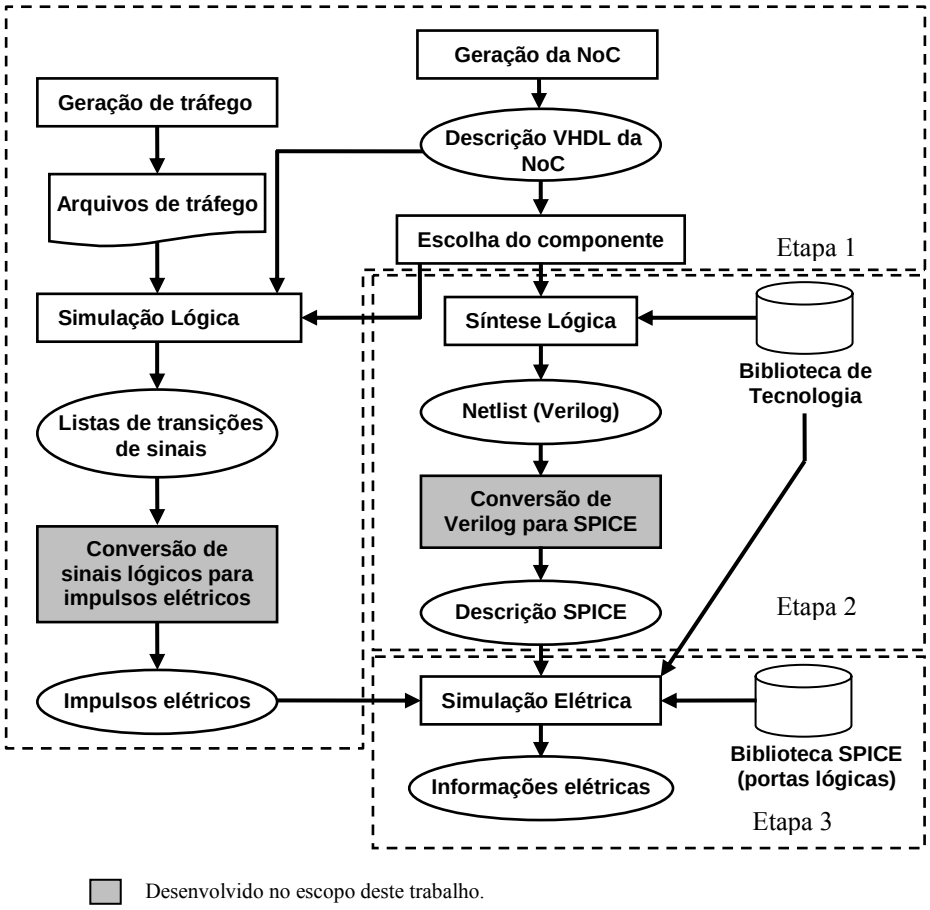


Figura 4.1: Fluxo para aquisição dos parâmetros do modelo de potência.

4.1.1 Etapa 1

A primeira etapa tem como objetivos: (*i*) gerar a descrição VHDL do componente (módulo de hardware) para o qual se deseja estimar o consumo de potência; e (*ii*) gerar os estímulos elétricos para cada sinal de entrada deste componente. Estes estímulos são gerados em formato PWL e são usados posteriormente, na simulação SPICE (etapa 3).

Esta etapa inicia com a geração da NoC (sem esquemas de codificação) e dos arquivos de tráfego, ambos obtidos através da ferramenta MAIA (MELLO et al., 2005). Conforme detalhado na seção 2.9.1, esta ferramenta gera a descrição estrutural VHDL da rede Hermes de acordo com os parâmetros determinados pelo usuário. A ferramenta gera também os arquivos de tráfego de entrada que alimentam a NoC através das portas locais dos roteadores, modelando o comportamento dos núcleos locais.

Um simulador VHDL (neste trabalho foram utilizados os simuladores *Modelsim* e *Active HDL*) lê os arquivos gerados pela feramenta MAIA, contendo os pacotes que caracterizam o tráfego, e aplica sinais de entrada à NoC realizando a simulação da mesma com este tráfego. Ao atingir as portas locais dos roteadores de destino, os pacotes são armazenados em arquivos de saída da NoC. Desta forma, é possível comparar os pacotes recebidos nos destinos com os pacotes enviados pelos roteadores de origem, assegurando o funcionamento correto da NoC.

O simulador fornece acesso ao comportamento dos vários conjuntos de sinais que interconectam os roteadores da NoC entre si, bem como dos sinais que interconectam os

componentes internos dos roteadores, ou seja, buffers e lógica de controle. Durante a simulação, o usuário pode escolher manualmente um componente desejado (como, por exemplo, o buffer de entrada da porta *Local* do roteador com endereço "00") e verificar o comportamento dos sinais de entrada e de saída deste componente, de acordo com o tráfego de entrada. Nesta etapa, são produzidas as listas de sinais de entrada, como a ilustrada na Figura 4.2, que armazenam estes sinais e suas transições ao longo do tempo. O próximo passo é converter estas listas para estímulos elétricos em formato PWL (Figura 4.3). Esta conversão é feita através de um programa desenvolvido no escopo deste trabalho.

ns	delta	clock	reset	rx(4)	rx(3)	rx(2)	rx(1)	rx(0)	ack_tx(4)	ack_tx(3)	ack_tx(2)	ack_tx(1)	ack_tx(0)
0	+3	1	1	0	0	0	0	0	0	0	0	0	0
10	+0	0	1	0	0	0	0	0	0	0	0	0	0
15	+0	0	0	0	0	0	0	0	0	0	0	0	0
20	+5	1	0	1	0	0	0	0	0	0	0	0	0
30	+0	0	0	1	0	0	0	0	0	0	0	0	0
40	+5	1	0	0	0	0	0	0	0	0	0	0	0
50	+0	0	0	0	0	0	0	0	0	0	0	0	0
60	+5	1	0	1	0	0	0	0	0	0	0	0	0
70	+0	0	0	1	0	0	0	0	0	0	0	0	0
80	+5	1	0	0	0	0	0	0	0	0	0	0	0
90	+0	0	0	0	0	0	0	0	0	0	0	0	0
100	+5	1	0	1	0	0	0	0	0	0	0	0	0
110	+0	0	0	1	0	0	0	0	0	0	0	0	0
120	+5	1	0	0	0	0	0	0	0	0	0	0	0
130	+0	0	0	0	0	0	0	0	0	0	0	0	0
140	+5	1	0	1	0	0	0	0	0	0	0	0	0
150	+0	0	0	1	0	0	0	0	0	0	0	0	0
160	+7	1	0	1	0	0	0	1	0	0	1	0	0

Figura 4.2: Lista de sinais de entrada do módulo escolhido e suas transições ao longo do tempo.

```
.Vclock clock 0 pwl( 0n 3.3 8.0n 3.3   10n 0.0 18.0n 0.0   20n 3.3 28.0n 3.3
+  30n 0.0 38.0n 0.0   40n 3.3 48.0n 3.3   50n 0.0 58.0n 0.0   60n 3.3 68.0n 3.3
+  70n 0.0 78.0n 0.0   80n 3.3 88.0n 3.3   90n 0.0 98.0n 0.0   100n 3.3 108.0n 3.3 )

Vdata_ack data_ack 0 pwl( 0n 0.0 8.0n 0.0   10n 0.0 18.0n 0.0   20n 0.0 28.0n 0.0
+  30n 0.0 38.0n 0.0   40n 0.0 48.0n 0.0   50n 0.0 58.0n 0.0   60n 0.0 68.0n 0.0
+  70n 0.0 78.0n 0.0   80n 0.0 88.0n 0.0   90n 0.0 98.0n 0.0   100n 0.0 108.0n 0.0 )

Vdata_in_31_ data_in_31_ 0 pwl( 0n 0.0 8.0n 0.0   10n 0.0 18.0n 0.0   20n 0.0 28.0n 0.0
+  30n 0.0 38.0n 0.0   40n 0.0 48.0n 0.0   50n 3.3 58.0n 3.3   60n 3.3 68.0n 3.3
+  70n 0.0 78.0n 0.0   80n 0.0 88.0n 0.0   90n 0.0 98.0n 0.0   100n 0.0 108.0n 0.0 )

Vreset reset 0 pwl( 0n 3.3 8.0n 3.3   10n 3.3 18.0n 3.3   20n 0.0 28.0n 0.0
+  30n 0.0 38.0n 0.0   40n 0.0 48.0n 0.0   50n 0.0 58.0n 0.0   60n 0.0 68.0n 0.0
+  70n 0.0 78.0n 0.0   80n 0.0 88.0n 0.0   90n 0.0 98.0n 0.0   100n 0.0 108.0n 0.0 )
```

Figura 4.3: Estímulos elétricos em formato PWL.

4.1.2 Etapa 2

Esta etapa tem como objetivo gerar a descrição SPICE do componente escolhido na etapa 1, o qual será simulado eletricamente na etapa 3, resultando nas estimativas de potência do mesmo.

Na etapa 2 o módulo escolhido é primeiramente sintetizado através da ferramenta *LeonardoSpectrum*, utilizando uma biblioteca de células para uma tecnologia específica. Neste trabalho foi utilizada a biblioteca TSMC 0.35µm. A síntese lógica é feita com base em uma lista de 46 portas lógicas implementadas em uma outra biblioteca (a de portas lógicas descritas em SPICE), que será utilizada na simulação elétrica da etapa 3. Isto evita que a ferramenta de síntese utilize uma porta não modelada em SPICE. A Figura 4.4 apresenta uma lista com os nomes destas 46 portas lógicas.

A ferramenta *LeonardoSpectrum* produz uma lista de conexões (*netlist*) HDL que é, em seguida, convertida para uma *netlist* SPICE através de um outro conversor também desenvolvido no escopo deste trabalho. A Figura 4.5 mostra um trecho da *netlist* HDL de um buffer Hermes, e a Figura 4.6 mostra o mesmo trecho após convertido para *netlist* SPICE.

A seção 4.2 apresenta os resultados de área dos módulos da NoC e dos esquemas de codificação.

Inv	Nand9	Nor5	Or9
Trg	And2	Nor6	Xor2
TriState	And3	Nor7	DffReset
Buffer	And4	Nor8	DffSet
Mux2to1	And5	Nor9	nand3
Nand2	And6	Or2	nxor2
Nand3	And7	Or3	latch
Nand4	And8	Or4	dff
Nand5	And9	Or5	dffclear
Nand6	Nor2	Or6	dffset_P
Nand7	Nor3	Or7	
Nand8	Nor4	Or8	

Figura 4.4: Portas lógicas implementadas na biblioteca SPICE.

```
nand04 ix2105 (.Y (data_0_), .A0 (nx5498), .A1 (nx6865), .A2 (nx6871), .A3 (nx6877)) ;
inv01 ix5503 (.Y (nx5502), .A (data_in_0_)) ;
dff ix5462 (.Q (\$dummy0), .QB (nx5504), .D (nx5459), .CLK (clock_rx)) ;
nand03 ix5507 (.Y (nx5506), .A0 (nx60), .A1 (nx7134), .A2 (nx7130)) ;
nor02 ix61 (.Y (nx60), .A0 (reset), .A1 (nx5509)) ;
nand02 ix5510 (.Y (nx5509), .A0 (rx), .A1 (credit_o)) ;
dffs ix2045 (.Q (credit_o), .QB (\$dummy1), .D (nx2042), .CLK (NOT_clock_rx), .S
(NOT_reset)) ;
nor02 ix2043 (.Y (nx2042), .A0 (nx5513), .A1 (nx6858)) ;
nor02 ix5514 (.Y (nx5513), .A0 (nx1978), .A1 (nx2036)) ;
mux21 ix3860 (.Y (nx3859), .A0 (nx6982), .A1 (nx7130), .S0 (nx5509)) ;
dffr ix3862 (.Q (\$dummy2), .QB (nx5519), .D (nx3859), .CLK (clock_rx), .R (reset)) ;
```

Figura 4.5: Trecho da *netlist* HDL de um buffer Hermes.

```
X0   data_0_      nx5498  nx6865    nx6871  nx6877  VCC  0  nand4
X1   nx5502  data_in_0_  VCC  0  inv
X2   __dummy0  nx5504  nx5459  clock_rx  VCC  0  dff
X3   nx5506  nx60  nx7134  nx7130  VCC  0  nand3
X4   nx60  reset  nx5509  VCC  0  nor2
X5   nx5509  rx  credit_o  VCC  0  nand2
X6   credit_o  __dummy1  nx2042  NOT_clock_rx  NOT_reset  VCC  0  dffset
X7   nx2042  nx5513  nx6858  VCC  0  nor2
X8   nx5513  nx1978  nx2036  VCC  0  nor2
X9   nx3859  nx6982  nx7130  nx5509  VCC  0  mux2to1
X10  __dummy2  nx5519  nx3859  clock_rx  reset  VCC  0  dffclear
```

Figura 4.6: Trecho da *netlist* SPICE de um buffer Hermes.

4.1.3 Etapa 3

A etapa 3 tem por objetivo produzir as informações elétricas relacionadas ao consumo de potência do módulo escolhido, de acordo com cada tipo de tráfego simulado. Esta etapa consiste na simulação SPICE do componente escolhido. O simulador SPICE recebe como entradas a *netlist* SPICE do módulo, os estímulos elétricos de entrada produzidos na primeira etapa e uma biblioteca com portas lógicas descritas em SPICE. A Figura 4.7 mostra um trecho desta biblioteca, descrevendo três

subcircuitos: *Inversor*, *TransmissionGate* e *Tri-State*. O circuito *Tri-State* é composto por um um *Inversor* e um *TransmissionGate*.

As informações elétricas resultante desta etapa são utilizadas na construção de um macromodelo de consumo de potência da NoC (ou de seus componentes internos) para um determinado tráfego.

```
.subckt Inversor out in Vcc 0
MP1  out in Vcc Vcc MODP L=0.35U W=5.0U AD=10.0P AS=10.0P PD=9.0U PS=9.0U
MN2  out in   0   0 MODN L=0.35U W=2.0U AD= 4.0P AS= 4.0P PD=6.0U PS=6.0U
.ends Inversor

.subckt TransmissionGate out in control notControl Vcc 0
M1 in  control    out   0 MODN L=0.35U W=2.0U AD= 4.0P AS= 4.0P PD=6.0U PS=6.0U
M2 in  notControl out Vcc MODP L=0.35U W=5.0U AD=10.0P AS=10.0P PD=9.0U PS=9.0U
.ends TransmissionGate

.subckt TriState out in control Vcc   0
X1 notControl control                    Vcc 0 Inversor
X2 out_1           in control notControl Vcc 0 TransmissionGate
.ends TriState
```

Figura 4.7: Porta lógica *Tri-State* no formato SPICE. A porta é composta pelos subcircuitos *Inversor* e *TransmissionGate*.

4.2 Resultados de Área dos Módulos

Esta seção apresenta os resultados de área, em termos de portas-lógicas e transistores, dos diferentes módulos da NoC e dos esquemas de codificação com diferentes configurações. A Tabela 4.1 mostra os resultados de área dos módulos de uma NoC Hermes com diferentes configurações de profundidade de buffer e largura de *flit*. A Tabela 4.2 apresenta os resultados de área dos módulos da NoCs Hermes com larguras de *flit* iguais a 9, 17, 18 e 36 bits, todas adaptadas ao esquema *Bus-Invert* com diferentes larguras de *flit*. Como se pode observar, os módulos *Switch Control* com 17 e com 18 bits possuem a mesma quantidade de portas lógicas e de transistores. Isto ocorre devido ao fato de que a lógica destes dois módulos é a mesma, sendo que no de 17 bits existe 1 sinal de inversão que controla 16 inversores, enquanto que o de 18 bits possui 2 sinais de controle que controlam, cada um deles, 8 inversores.

A Tabela 4.3 mostra os mesmos resultados para os módulos dos esquemas de codificação implementados neste trabalho, também com diferentes larguras de *flit*. O esquema *Adaptive Encoding* foi implementado somente com 8 bits devido à sua elevada complexidade, como se pode observar na Tabela 4.3, e elevado consumo de potência, como será ilustrado na seção 4.4. Na Tabela 4.3 também se pode observar que os módulos *decoder* do esquema *Bus-Invert* de 16 bits com 1 e com 2 clusters possuem a mesma área. Isto ocorre devido pelo mesmo motivo do que nos módulos *Switch Control* com 17 e com 18 bits, comentado anteriormente. Por fim, a Tabela 4.4 mostra os resultados de área dos roteadores da rede Hermes, de acordo com suas configurações de largura de *flit* e profundidade de buffers, e de acordo com a sua posição na NoC. Como se pode observar na Figura 4.8, os roteadores da periferia de uma NoC com topologia mesh possuem um número menor de portas de entrada e de saída. Isto faz com que estes roteadores sejam consituídos de um número menor de buffers internos, reduzindo sua

área. Na Figura 4.8 são identificados os buffers de entrada das portas *North* (N), *South* (S), *East* (E), *West* (W) e *Local* (L) de cada roteador, quando existirem.

Tabela 4.1: Resultados de área dos módulos de uma NoC Hermes com diferentes configurações.

Largura de *Flit*	Componente	Portas lógicas	Transistores
8 bits	Buffer 8 x 4	568	2714
	Buffer 8 x 8	904	4264
	Buffer 8 x 16	1535	7160
	Switch Control 8	622	2840
16 bits	Buffer 16 x 4	976	4578
	Buffer 16 x 8	1623	7518
	Buffer 16 x 16	2837	13038
	Switch Control 16	678	3152
32 bits	Buffer 32 x 4	1780	8332
	Buffer 32 x 8	3009	13892
	Buffer 32 x 16	5371	24566
	Switch Control 32	836	4176

Tabela 4.2: Resultados de área dos módulos de uma NoC Hermes com diferentes configurações alteradas para o uso com o esquema Bus-Invert.

Largura de *Flit*	Componente	Portas lógicas	Transistores
9 bits	Buffer 9 x 16	1683	7952
	Switch Control 9	643	3004
17 bits	Buffer 17 x 16	3010	14024
	Switch Control 17	700	3360
18 bits	Buffer 18 x 16	3147	14650
	Switch Control 18	700	3360
36 bits	Buffer 36 x 16	6006	27842
	Switch Control 36	915	4308

Tabela 4.3: Resultados de área dos módulos de codificação e decodificação de diferentes esquemas de codificação.

Esquemas	Componente	Portas lógicas	Transistores
Adaptive Encoding 8 bits	Encoder	2031	9482
	Decoder	2045	9512
Bus-Invert 8 bits	Encoder	217	1128
	Decoder	73	446
Transition 8 bits	Encoder	300	1534
	Decoder	313	1572
Gray 8 bits	Encoder	228	1240
	Decoder	227	1238
T-Bus-invert 8 bits	Encoder	670	3242
	Decoder	499	2446
Bus-Invert 16 bits – 1 cluster	Encoder	448	2712
	Decoder	145	888
Bus-Invert 16 bits – 2 clusters	Encoder	416	2512
	Decoder	145	888
Transition 16 bits	Encoder	574	2914
	Decoder	600	3002
Gray 16 bits	Encoder	436	2366
	Decoder	436	2366
T-Bus-invert 16 bits	Encoder	1325	6428
	Decoder	991	4852
Bus-Invert 32 bits – 4 clusters	Encoder	867	4512
	Decoder	287	1766
Transition 32 bits	Encoder	1115	5664
	Decoder	1165	5832
Gray 32 bits	Encoder	852	4630
	Decoder	852	4632
T-Bus-invert 32 bits	Encoder	2643	12820
	Decoder	1975	9666

Tabela 4.4: Resultados de área dos roteadores de uma NoC Hermes com diferentes configurações e diferentes números de portas, dependendo de sua posição na NoC de topologia mesh.

Largura de *Flit*	Profundidade dos Buffers	Número de Portas do Roteador	Portas lógicas	Transistores
8 bits	4 palavras	3	2465	11564
		4	3195	15064
		5	3955	18656
	8 palavras	3	3457	16112
		4	4521	21134
		5	5666	26414
	16 palavras	3	5339	24738
		4	7043	32650
		5	9324	42700
16 bits	4 palavras	3	3924	18132
		4	5127	23782
		5	6472	30016
	8 palavras	3	5829	26866
		4	7706	35672
		5	10036	44560
	16 palavras	3	10182	45636
		4	13416	60062
		5	16568	73948
32 bits	4 palavras	3	6791	31350
		4	9256	40228
		5	11189	48228
	8 palavras	3	11261	50128
		4	14817	65746
		5	18145	80136
	16 palavras	3	18927	84304
		4	25039	111318
		5	30923	137100

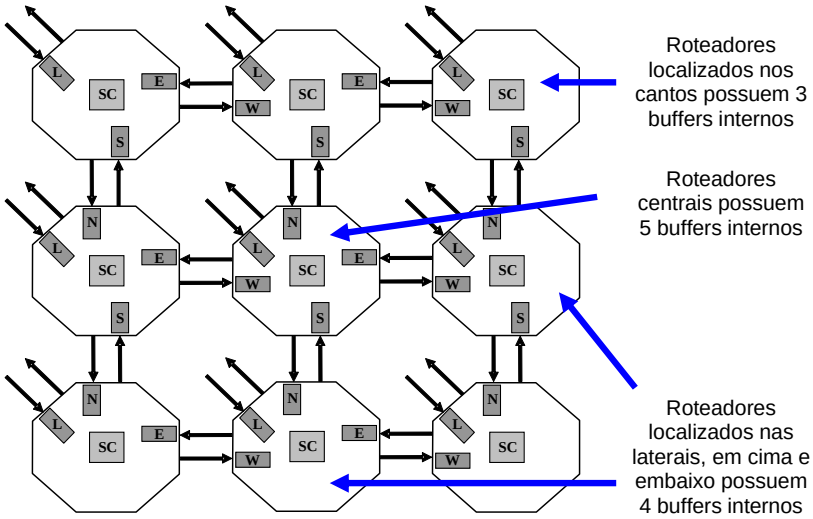


Figura 4.8: Roteadores com diferentes quantidades de buffers em uma NoC com topologia mesh.

4.3 Definição do Modelo

Esta seção apresenta a definição do modelo de consumo de potência desenvolvido e utilizado neste trabalho. Neste modelo são utilizados vários parâmetros de potência, indicando o consumo médio em diferentes componentes da NoC e dos esquemas de codificação.

O parâmetro *APH* (*Average Power per Hop*), por exemplo, é utilizado para indicar o consumo médio de potência em um único *Hop* percorrido por um pacote transmitido através da NoC. O *APH* pode ser dividido em três componentes: *(i)* potência média consumida em um roteador composto de buffers e lógica interna para roteamento e chaveamento (*APR – Average Power per Router*); *(ii)* potência média consumida no canal de comunicação entre roteadores (APL_R); e *(iii)* potência média consumida no canal de comunicação entre um roteador e seu núcleo local (APL_L). A Equação (7) representa a potência média consumida por um pacote transmitido através de um roteador, um canal de comunicação entre roteadores e um canal de comunicação local.

$$APH = APR + APL_R + APL_L \quad (7)$$

Além disso, os resultados obtidos mostram que uma avaliação melhor da potência média consumida no roteador pode ser feita dividindo-a em: *(i)* potência média do buffer (APB_F) e *(ii)* potência média do módulo de controle (APS_C). Isto se deve ao fato de que o efeito das transições de sinais sobre a potência consumida no controle é muito menor do que o efeito sobre a potência consumida no buffer. A Figura 4.9 ilustra este efeito em um buffer com profundidade de 16 palavras e na lógica de controle de um roteador Hermes com largura de *flit* igual a 8 bits. O gráfico mostra a potência média (em mW) consumida em função da quantidade de transições de sinais em um pacote com 128 *flits* (100% = 127 transições em cada um dos 8 fios do canal de comunicação). Na figura é possível observar que o consumo de potência aumenta linearmente com o aumento das transições.

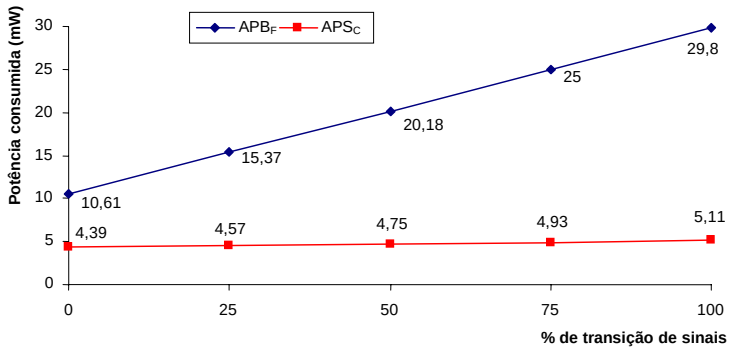


Figura 4.9: Análise do efeito da transição de sinais sobre a potência média consumida em um buffer de 16 palavras e na lógica de controle de um roteador Hermes com largura de flit igual a 8 bits. Dados obtidos através da simulação SPICE (tecnologia CMOS TSMC 0.35μ).

Em arquiteturas regulares baseadas em *tiles*, as dimensões do *tile* são próximas das dimensões médias dos núcleos. Além disso, as portas de entrada e saída do núcleo são posicionadas próximas à porta local do roteador (assumindo que em cada *tile* há um núcleo local, e não um subsistema com mais núcleos), fazendo com que o canal de comunicação local seja bem menor do que o canal de comunicação entre roteadores. Sendo assim, APL_L é bem menor que APL_R, apresentando um consumo médio de potência insignificante, mesmo no pior caso, como mostra a Figura 4.10.

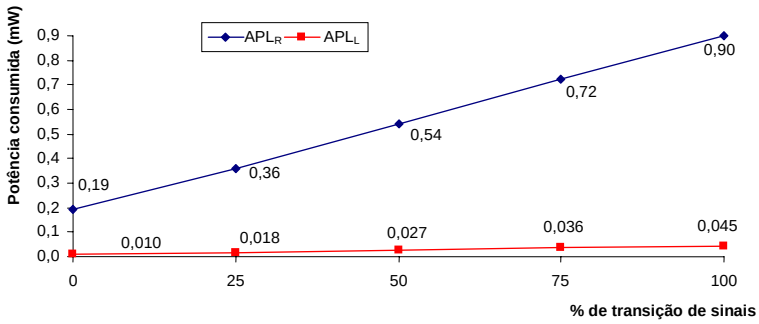


Figura 4.10: Análise do efeito das transições de sinais sobre a potência média consumida nos canais de comunicação locais e entre roteadores. Cada tile tem dimensões de 5 mm x 5 mm e os canais de comunicação possuem largura de *flit* igual a 8 bits.

Com base nestes resultados, APL_L pode ser desconsiderado sem causar erros significativos no cálculo da potência dissipada. A Equação (8) computa a potência média consumida na comunicação entre um roteador *i* (no *tile* τ*i*) e um roteador *j* (no *tile* τ*j*), onde η corresponde ao número de roteadores que o pacote deve percorrer.

$$RRPij = \eta \times (APB_F + APS_C) + (\eta - 1) \times APL_R \quad (8)$$

Considerando uma abordagem que utilize um esquema de codificação de dados, dois novos parâmetros devem ser adicionados à Equação (8): *APE* e *APD* (potência média consumida nos módulos de codificação e decodificação, respectivamente). A Figura 4.11 mostra a análise do efeito das transições de sinais sobre a potência média consumida nos módulos de codificação e decodificação do esquema de codificação *Adaptive Encoding*, utilizado aqui como exemplo, ambos com largura de *flit* igual a 8 bits. Como se pode observar na figura, o consumo de potência também cresce linearmente nestes módulos, de acordo com o aumento da quantidade de transições de sinais (PALMA et al., 2006-b).

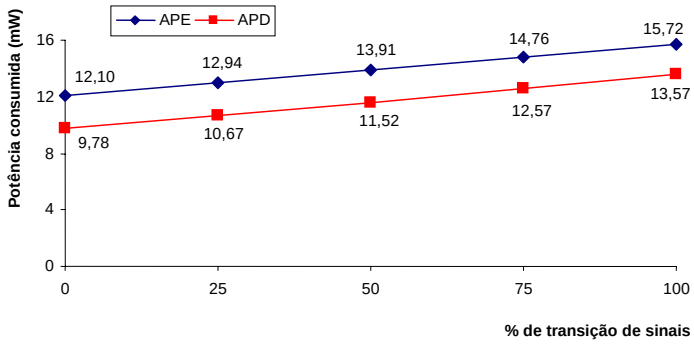


Figura 4.11: Análise do efeito das transições de sinais sobre a potência média consumida nos módulos de codificação e decodificação do esquema Adaptive Probability Encoding com largura de *flit* igual a 8 bits.

A Equação (9) computa a potência média consumida na comunicação entre o núcleo local do roteador localizado no *tile* τi e o núcleo local do roteador localizado no *tile* τj, passando por η roteadores e utilizando o esquema de codificação de dados.

$$CodedRRPij = APE + \eta \times (APB_F + APS_C) + (\eta - 1) \times APL_R + APD \quad (9)$$

Com base nas análises acima descritas, é possível construir macromodelos para cada um dos parâmetros do modelo proposto – APB_F, APS_C, *APE*, *APD* e APL_R – representando a potência média consumida nos diferentes módulos da NoC. Estes macromodelos serão apresentados na seção 4.5. A secao 4.4 mostra os gráficos de consumo de potência obtidos após a simulação SPICE de diferentes configurações da NoC e dos módulos de codificação e decodificação.

4.4 Gráficos de Consumo de Potência nos Componentes da NoC e nos Módulos de Diferentes Esquemas de Codificação

A Figura 4.12 mostra a análise do efeito da transição de sinais sobre o consumo médio de potência em buffers com 4, 8 e 16 palavras e na lógica de controle de um roteador Hermes com largura de flit igual a 8 bits. A Figura 4.13 ilustra a mesma análise utilizando um roteador Hermes com largura de flit igual a 16 bits (PALMA et al., 2007-b), enquanto que a Figura 4.14 ilustra esta análise em um roteador Hermes com largura de flit igual a 32 bits.

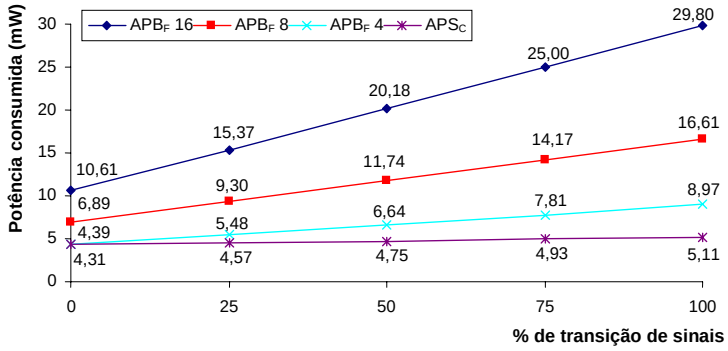


Figura 4.12: Análise do efeito da transição de sinais sobre o consumo médio de potência em buffers com 4, 8 e 16 palavras e na lógica de controle de um roteador Hermes com largura de *flit* igual a 8 bits.

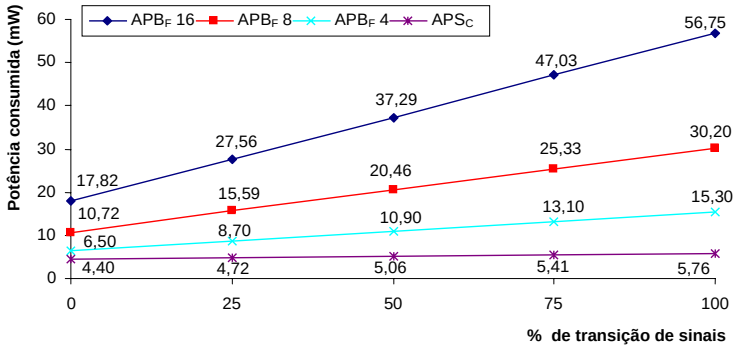


Figura 4.13: Análise do efeito da transição de sinais sobre o consumo médio de potência em buffers com 4, 8 e 16 palavras e na lógica de controle de um roteador Hermes com largura de *flit* igual a 16 bits.

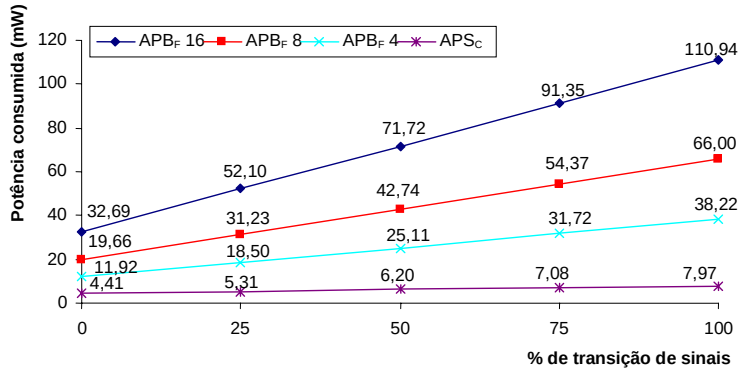


Figura 4.14: Análise do efeito da transição de sinais sobre o consumo médio de potência em buffers com 4, 8 e 16 palavras e na lógica de controle de um roteador Hermes com largura de flit igual a 32 bits.

A Figura 4.15 apresenta os resultados da análise do efeito da transição de sinais sobre o consumo médio de potência em canais de comunicação entre roteadores com larguras de *flit* iguais a 8, 16 e 32 bits, considerando tiles com dimensões 5 mm x 5 mm.

A Figura 4.16 mostra os resultados da mesma análise sobre canais de comunicação locais também com larguras de *flit* iguais 8, 16 e 32 bits.

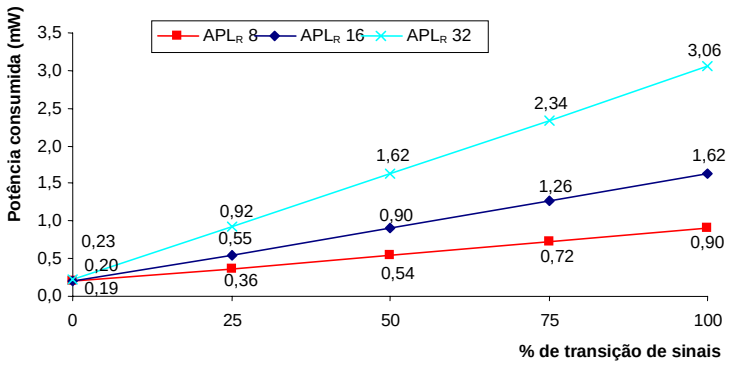


Figura 4.15: Análise do efeito da transição de sinais sobre o consumo médio de potência em canais entre roteadores com 8, 16 e 32 bits. As dimensões do tile são 5 mm x 5 mm.

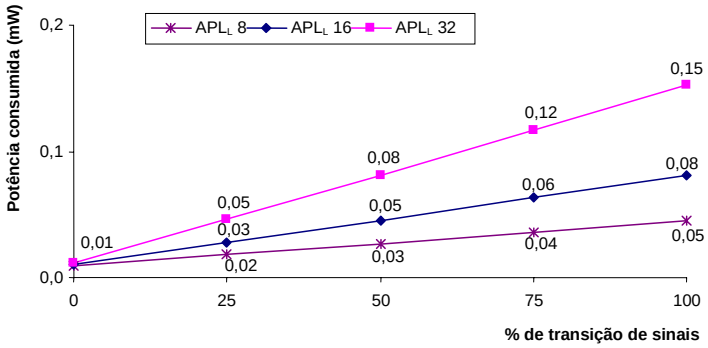


Figura 4.16: Análise do efeito da transição de sinais sobre o consumo médio de potência em canais locais com 8, 16 e 32 bits.

Conforme comentado anteriormente, também é necessário que se analise o consumo médio de potência nos módulos de codificação e decodificação dos diferentes esquemas apresentados neste trabalho. Desta forma pode-se avaliar corretamente o custo da codificação em termos de consumo extra de potência, frente à redução de potência na NoC, obtida com a redução da atividade de transição de sinais nos módulos da mesma. A Figura 4.17 apresenta esta análise sobre os módulos de codificação dos esquemas *Bus-Invert*, *Gray*, *Adaptive Probability Encoding*, *Transition* e *T-Bus-Invert*, ambos utilizados em uma NoC com largura de flit igual a 8 bits. A Figura 4.18 apresenta a análise sobre os módulos de decodificação destes mesmos esquemas de codificação, também com largura de *flit* igual a 8 bits (PALMA et al., 2007-b).

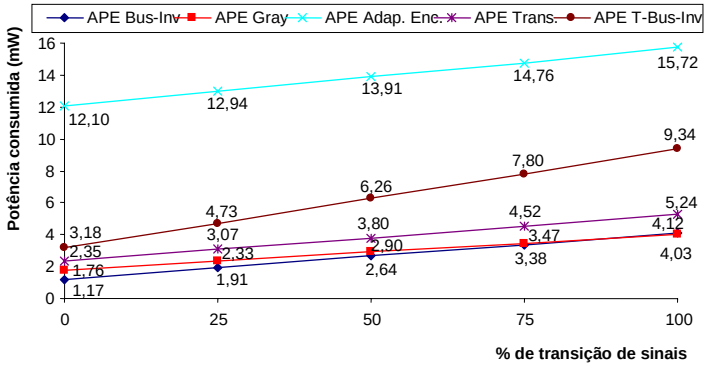


Figura 4.17: Análise do efeito da transição de sinais sobre o consumo médio de potência em módulos de codificação de diferentes esquemas, utilizando largura de flit igual a 8 bits.

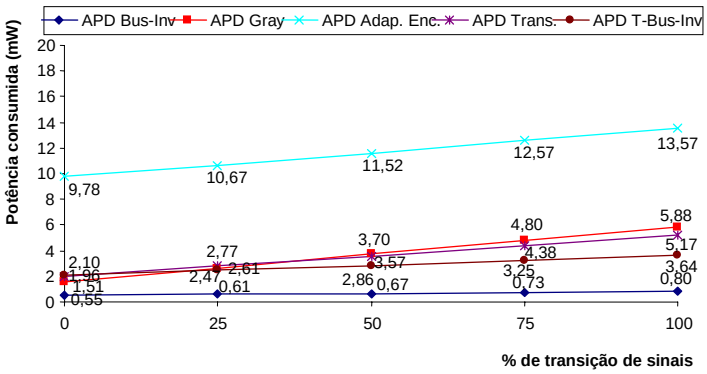


Figura 4.18: Análise do efeito da transição de sinais sobre o consumo médio de potência em módulos de decodificação de diferentes esquemas, utilizando largura de flit igual a 8 bits.

Como se pode observar na Figura 4.17 e na Figura 4.18, os módulos de codificação e de decodificação do esquema *Adaptive Probability Encoding* apresentam um consumo médio de potência elevado, comparado aos módulos de codificação e de decodificação dos outros esquemas implementados neste trabalho. Isto deve-se ao fato de que este esquema possui contadores individuais para cada linha de comunicação, utilizados para extrair as informações estatísticas do tráfego, as quais definem a função de codificação a ser utilizada a cada ciclo de clock. Estes contadores tornam estes módulos grandes em termos de portas lógicas e com alto consumo de potência.

Sendo os contadores individuais para cada linha de comunicação, o tamanho e o consumo de potência de um codificador ou decodificador do esquema *Adaptive Probability Encoding* com 16 ou 32 bits seria bastante elevado. Por este motivo optou-se pela não implementação deste esquema de codificação em NoCs com largura de *flit* maior do que 8 bits.

Como se pode observar na Figura 4.19 e na Figura 4.20 uma nova versão do esquema Bus-Invert, com 2 clusters, foi implementado no lugar do esquema *Adaptive Probability Encoding* com 16 bits. Nestas figuras são mostradas as análises de duas abordagens utilizando o esquema Bus-Invert. A primeira abordagem apresenta o esquema *Bus-Invert* com 1 *cluster*, ou seja, utilizando 1 bit de controle e executando a

função de maioria sobre os 16 bits do canal de comunicação. Neste caso a inversão é feita quando mais de 8 bits são diferentes, conforme descrito na seção 3.5. A segunda abordagem apresenta o esquema *Bus-Invert* com 2 *clusters*, ou seja, utilizando 2 bits de controle e executando a função de maioria separadamente em cada um dos dois agrupamentos de 8 bits do canal de comunicação. Neste caso a inversão é feita quando mais de 4 bits são diferentes no agrupamento de 8 bits.

A Figura 4.19 apresenta os resultados da análise do efeito da transição de sinais sobre o consumo médio de potência nos módulos de codificação dos esquemas *Bus-Invert* com 1 *cluster*, *Bus-Invert* com 2 *clusters*, *Gray*, *Transition* e *T-Bus-Invert*, ambos utilizados em uma NoC com largura de *flit* igual a 16 bits. Conforme ilustrado nesta figura, o codificador que apresenta o consumo de potência mais elevado é o do esquema *T-Bus-Invert*. Isto acontece pelo fato de que este codificador possui uma máquina-de-estados responsável pelo controle das combinações entre sinais de entrada e valores armazenados no buffer temporário, criando agrupamentos diferentes em cada um dos 8 estados possíveis, conforme descrito na seção 3.7.

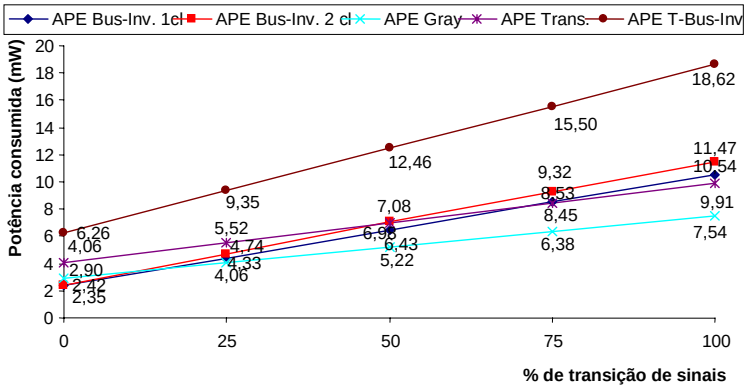


Figura 4.19: Análise do efeito da transição de sinais sobre o consumo médio de potência em módulos de codificação de diferentes esquemas, utilizando largura de flit igual a 16 bits.

A Figura 4.20 apresenta os resultados da mesma análise sobre os módulos de decodificação destes mesmos esquemas de codificação, também com largura de *flit* igual a 16 bits. O decodificador com consumo de potência mais elevado é o do esquema *Gray*, devido ao fato de que cada bit a ser decodificado depende da decodificação anterior, o que cria um aumento do caminho crítico, bem como da complexidade deste módulo, como explicado na seção 3.1. Com *flits* de 8 bits a complexidade do decodificador não é significativa. Porém, com 16 bits ou mais, ela torna-se significativa, fazendo com que o consumo de potência do módulo seja bem mais elevado do que os decodificadores dos outros esquemas.

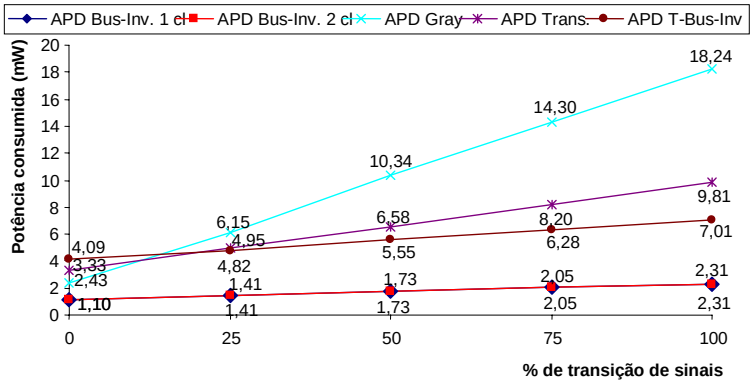


Figura 4.20: Análise do efeito da transição de sinais sobre o consumo médio de potência em módulos de decodificação de diferentes esquemas, utilizando largura de *flit* igual a 16 bits.

Outro detalhe a ser observado é o fato de que o consumo de potência dos decodificadores dos esquemas *Bus-Invert* com 1 *cluster* e com 2 *clusters* é exatamente o mesmo. Isto acontece por que as duas versões possuem a mesma estrutura e a mesma quantidade de portas lógicas. A única diferença é a quantidade de sinais de entrada. Na versão com *flit* de 17 bits, os 16 inversores responsáveis por inverter o dado quando necessário são controlados por uma única entrada (1 bit de controle). Já na versão com *flit* de 18 bits, estes 16 inversores são separados em dois grupos de 8, cada grupo controlado por uma das duas entradas extras (2 bits de controle) (PALMA et al., 2007-b). A Figura 4.21 e a Figura 4.22 mostram a análise do efeito da transição de sinais sobre o consumo médio de potência em módulos de codificação e de decodificação de diferentes esquemas, utilizando largura de *flit* igual a 32 bits. Para a NoC com largura de *flit* igual a 32 bits foram implementados os esquemas *Bus-Invert* com 4 *clusters* (4 agrupamentos de 8 bits e 4 bits de controle), *Gray*, *Transition* e *T-Bus-Invert*. Assim como na implementação com *flits* de 16 bits, o codificador do esquema *T-Bus-Invert* foi o que apresentou o consumo de potência mais elevado, bem como o decodificador do esquema *Gray*, utilizando *flits* de 32 bits.

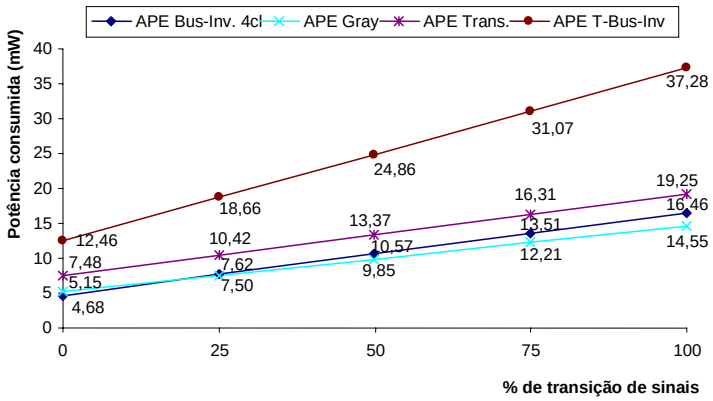


Figura 4.21: Análise do efeito da transição de sinais sobre o consumo médio de potência em módulos de codificação de diferentes esquemas, utilizando largura de *flit* igual a 32 bits.

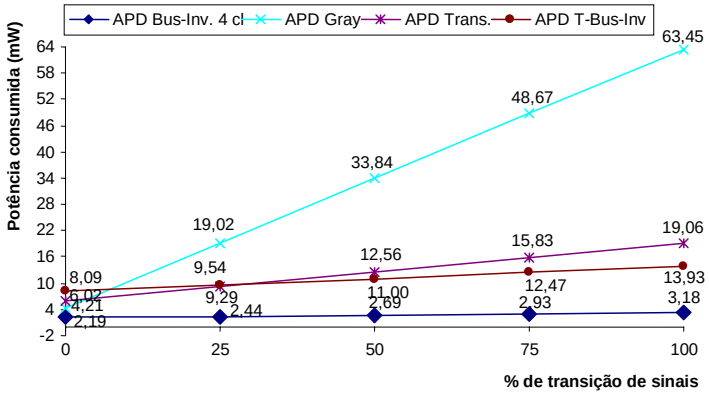


Figura 4.22: Análise do efeito da transição de sinais sobre o consumo médio de potência em módulos de decodificação de diferentes esquemas, utilizando largura de *flit* igual a 32 bits.

Consideremos agora uma NoC que utiliza o esquema de codificação *Bus-Invert*. Conforme descrito anteriormente, este esquema utiliza 1 ou mais bits de controle em cada *flit* transmitido pela NoC, ou seja, ele insere redundância espacial no dado transmitido. Isto implica em 1 ou mais fios a mais nos módulos da NoC (buffers e lógica de controle), bem como nos canais de comunicação locais e nos canais entre roteadores. Desta forma, é necessário fazer modificações no projeto da NoC, adaptando-a a esta abordagem, o que não acontece com as outras abordagens que utilizam os demais esquemas de codificação implementados neste trabalho.

As NoCs modificadas foram validadas por simulação e, em seguida, seus módulos foram sintetizados e analisados eletricamente, resultando em uma nova avaliação de consumo de potência. Em outras palavras, os módulos foram re-submetidos ao processo discutido na seção 4.1.

A Figura 4.23 ilustra o efeito da inserção de 1 bit extra em um buffer Hermes com largura de *flit* igual a 8 bits, comparando sua versão normal e sua versão adaptada ao esquema *Bus-Invert*, com largura de *flit* igual a 9 bits. Ambas as versões do buffer apresentam profundidade de 16 palavras. Similarmente, a Figura 4.24 ilustra este efeito na lógica de controle de um roteador Hermes com larguras de *flit* igual a 8 e 9 bits. A avaliação do consumo de potência também foi feita para os canais de comunicação locais e entre roteadores da NoC modificada. A Figura 4.25 mostra a avaliação nos canais entre roteadores. Nos canais locais, a inserção de 1 bit extra praticamente não alterou o consumo de potência, que permaneceu bem menor do que nos canais entre roteadores.

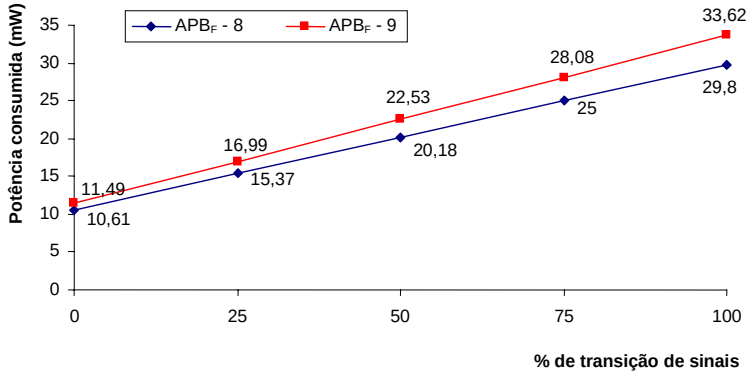


Figura 4.23: Análise do efeito da inserção de 1 bit extra em um buffer de um roteador Hermes com largura de *flit* igual a 8 bits, comparando sua versão normal e sua versão adaptada ao esquema *Bus-Invert*, com largura de *flit* igual a 9 bits.

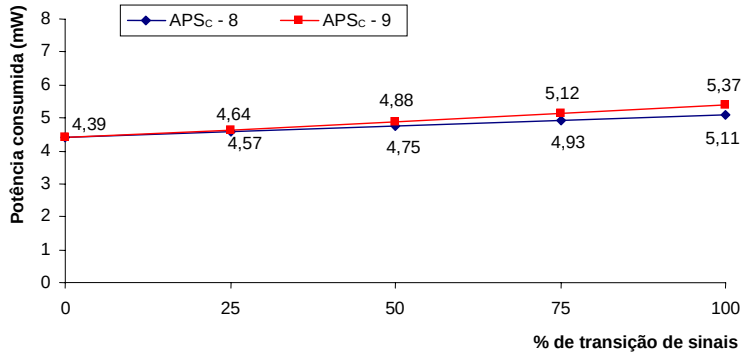


Figura 4.24: Análise do efeito da inserção de 1 bit extra na lógica de controle de um roteador Hermes com largura de *flit* igual a 8 bits, comparando sua versão normal e sua versão adaptada ao esquema *Bus-Invert*, com largura de *flit* igual a 9 bits.

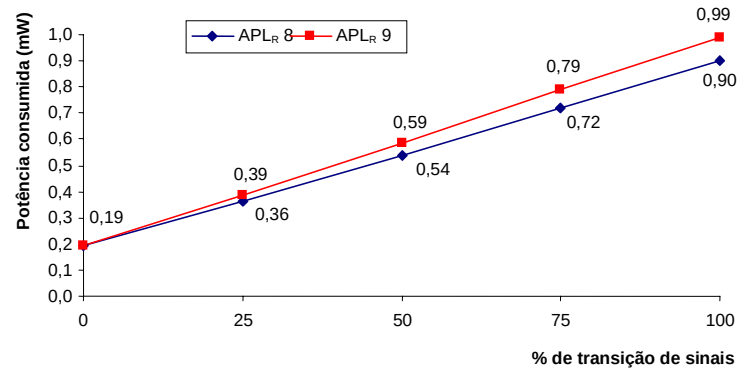


Figura 4.25: Análise do efeito da inserção de 1 bit extra no canal de comunicação entre roteadores com largura de flit igual a 8 bits, comparando sua versão normal e sua versão adaptada ao esquema Bus-Invert, com largura de *flit* igual a 9 bits.

Conforme mostra a Figura 4.23, a inserção de 1 bit a mais nos módulos da NoC com largura de *flit* igual a 8 bits, tem como conseqüência um aumento de 12% no consumo

de potência do buffer, no pior caso, com 100% de transição de sinais. A Figura 4.24 e a Figura 4.25 mostram que o aumento máximo foi de 5% na lógica de controle e de 10% no canal de comunicação entre roteadores.

A Figura 4.26 apresenta a análise feita sobre a variação do consumo de potência em um buffer Hermes, com largura de *flit* igual a 16 bits, com a inserção de 1 e 2 bits extras, adaptando o buffer ao esquema *Bus-Invert* com 1 e com 2 *clusters*. Estas duas novas versões são comparadas com a versão normal, ambas tendo profundidade de 16 palavras. A Figura 4.27 mostra esta mesma análise sobre a lógica de controle de um roteador Hermes, comparando a versão normal com as duas versões alteradas, com larguras de *flit* iguais a 17 e 18 bits. Nesta figura se pode observar que os consumos de potência nas versões da lógica de controle com *flits* de 17 e de 18 bits são iguais. Isto ocorre pelo mesmo motivo que nos módulos de decodificação dos esquemas *Bus-Invert* com 1 *cluster* e com 2 *clusters*. Ou seja, novamente os módulos com com *flits* de 17 e de 18 bits tem a mesma quantidade de portas-lógicas, diferindo apenas na quantidade de sinais de entrada. A Figura 4.28 mostra a análise feita nos canais entre roteadores, com 16, 17 e 18 bits (PALMA et al., 2007-b).

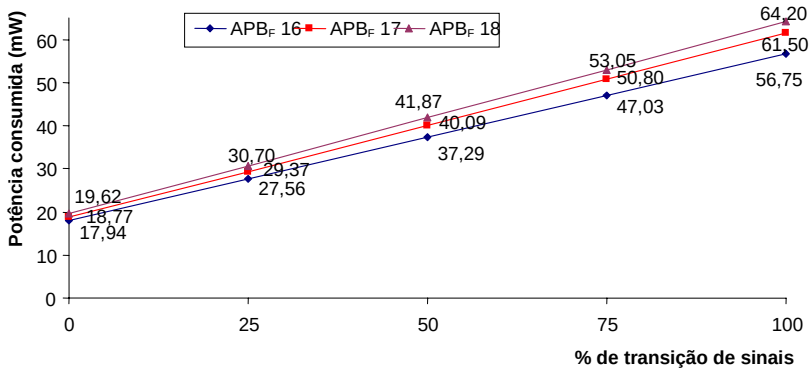


Figura 4.26: Análise do efeito da inserção de 1 e 2 bits extras em um buffer de um roteador Hermes com largura de flit igual a 16 bits, comparando sua versão normal com suas versões adaptadas ao esquema Bus-Invert, com larguras de *flit* iguais a 17 e 18 bits.

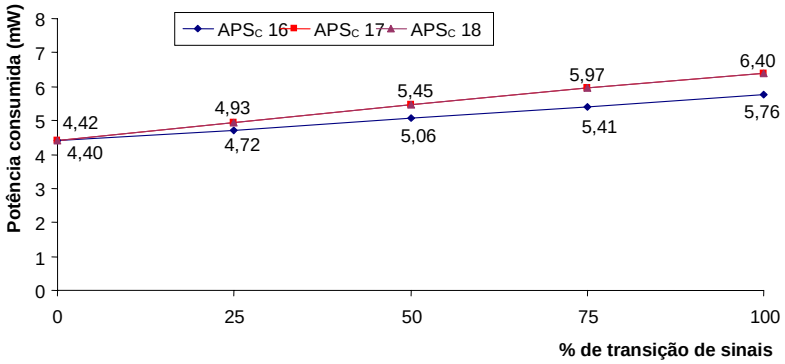


Figura 4.27: Análise do efeito da inserção de 1 e 2 bits extras na lógica de controle de um roteador Hermes com largura de *flit* igual a 16 bits, comparando sua versão normal com suas versões adaptadas ao esquema *Bus-Invert*, com larguras de *flit* iguais a 17 e 18 bits.

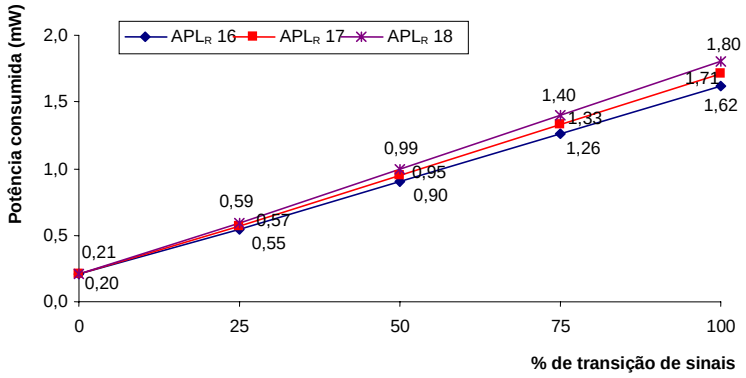


Figura 4.28: Análise do efeito da inserção de 1 e 2 bits extras no canal de comunicação entre roteadores com largura de flit igual a 16 bits, comparando sua versão normal com suas versões adaptadas ao esquema Bus-Invert, com larguras de *flit* iguais a 17 e 18 bits.

A inserção de 1 bit a mais nos módulos da NoC com largura de *flit* igual a 16 bits tem como conseqüência, no pior caso, um aumento de 8% no consumo de potência do buffer, 11% no consumo de potência da lógica de controle e 5,5% no consumo de potência do canal de comunicação entre roteadores. Já a inserção de 2 bits aumenta o consumo de potência no buffer em no máximo 13%, na lógica de controle em 11% e no canal de comunicação entre roteadores em 11%.

No caso da abordagem que utiliza o esquema *Bus-Invert* em uma NoC com largura de *flit* igual a 32 bits, somente foi implementada a versão deste esquema de codificação utilizando 4 clusters de 8 bits. A Figura 4.29 apresenta os resultados da análise da inserção de 4 bits extras em um buffer de um roteador Hermes com largura de *flit* igual a 32 bits, comparando a versão normal com a adaptada, com largura de *flit* igual a 36 bits. A mesma análise foi feita sobre a variação no consumo da lógica de controle deste mesmo roteador, como mostra a Figura 4.30, e sobre o canal de comunicação entre roteadores, como mostra a Figura 4.31.

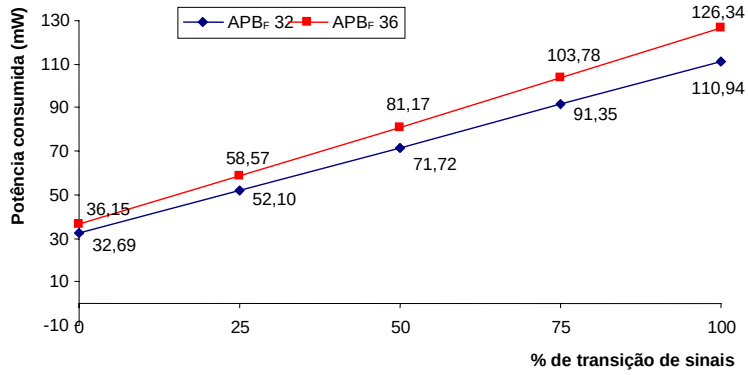


Figura 4.29: Análise do efeito da inserção de 4 bits extras em um buffer de um roteador Hermes com largura de flit igual a 32 bits, comparando sua versão normal com sua versão adaptada ao esquema Bus-Invert, com largura de *flit* igual a 36 bits.

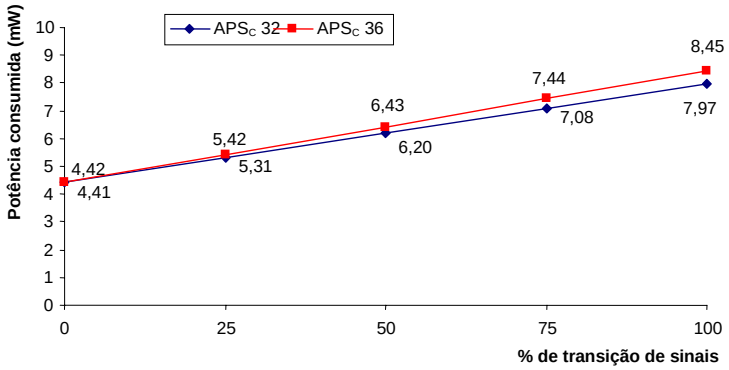


Figura 4.30: Análise do efeito da inserção de 4 bits extras na lógica de controle de um roteador Hermes com largura de *flit* igual a 32 bits, comparando sua versão normal e sua versão adaptada ao esquema *Bus-Invert*, com largura de *flit* igual a 36 bits.

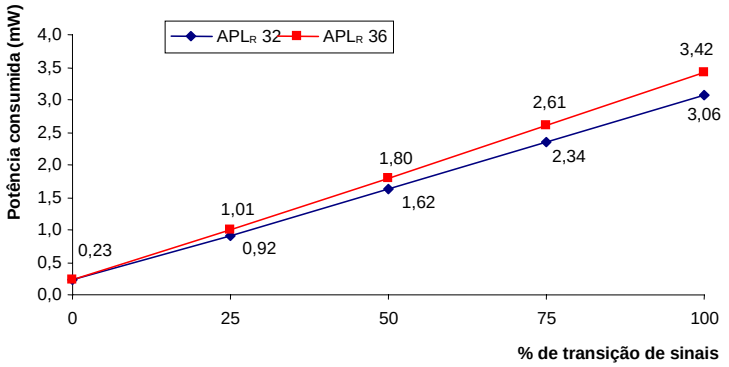


Figura 4.31: Análise do efeito da inserção de 4 bits extras no canal de comunicação entre roteadores com largura de flit igual a 32 bits, comparando sua versão normal e sua versão adaptada ao esquema Bus-Invert, com largura de *flit* igual a 36 bits.

O aumento máximo no consumo de potência, com a inserção de 4 bits, foi de 13,9% no buffer, 6% na lógica de controle e 11,7% no canal de comunicação entre roteadores.

4.5 Macromodelos de Consumo de Potência

Com base nos resultados obtidos após a simulação SPICE, é possível observar que todos os módulos (*buffer*, *control*, *encoder* e *decoder*), bem como os canais de comunicação, apresentam um consumo de potência mesmo quando a taxa de transição de sinais é de 0%. Isto acontece, nos módulos, devido aos chaveamentos de sinais internos e atualizações de estado, ativados pelo *clock*. Da mesma forma, nos canais de comunicação, mesmo quando se transmite somente ‘0’s nos dados do pacote, os sinais de *clock* e de controle de fluxo permanecem chaveando, consumindo potência. Este consumo é referido como P_0 na análise subseqüente. O restante da potência consumida cresce linearmente com a taxa de transição, até atingir o topo da rampa (P_1) que corresponde ao consumo máximo do módulo, quando a atividade de transição nos sinais de entrada é de 100%. A diferença entre P_1 e P_0 é referida como R. A Figura 4.32 ilustra a localização dos parâmetros P_0, R e P_1 no gráfico de consumo de potência de um buffer Hermes de 16 palavras e com largura de *flit* igual a 8 bits.

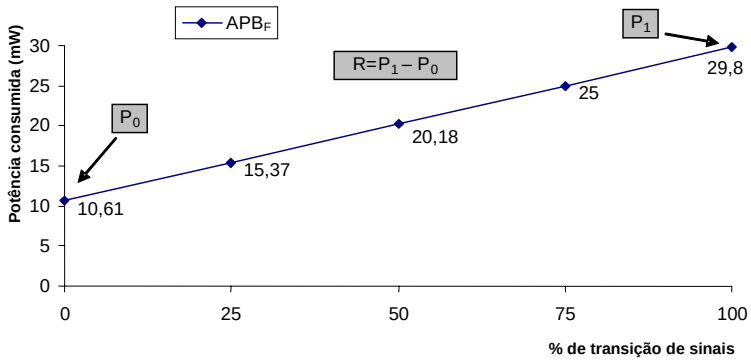


Figura 4.32: Parâmetros P_0 e R para o macromodelo de consumo de potência em um buffer de 16 palavras de um roteador Hermes com largura de *flit* igual a 8 bits.

A Tabela 4.5 apresenta os macromodelos lineares dos módulos de uma NoC Hermes com largura de *flit* igual a 8 bits. É importante observar que cada configuração do buffer, com profundidades diferentes (quantidade de palavras), possui seu próprio macromodelo, pois o consumo de potência é diferente em cada configuração. Já o módulo de controle e os canais de comunicação não são afetados pela configuração do buffer, apresentando um único macromodelo. Para calcular o consumo de potência de um determinado tráfego na NoC original basta utilizar os macromodelos abaixo, considerando a percentagem de chaveamento de bits ocasionada pelo tráfego. O método para realizar este cálculo será explicado no Capitulo 5.

Tabela 4.5: Macromodelos lineares dos módulos de uma NoC Hermes com largura de *flit* igual a 8 bits.

Módulo	P_0	R
Buffer (APB_F) – 4 palavras	4,31	4,58
Buffer (APB_F) – 8 palavras	6,89	9,72
Buffer (APB_F) – 16 palavras	10,61	19,19
Control (APS_C)	4,31	0,8
Canal entre roteadores (APL_R)	0,19	0,71

A Tabela 4.6 mostra os macromodelos dos módulos dos esquemas de codificação *Gray*, *Adaptive Probability Encoding*, *Transition* e *T-Bus-Invert*, todos utilizando largura de *flit* igual a 8 bits. Para utilizar os módulos de codificação e decodificação destes quatro esquemas de codificação em uma NoC, não é necessário que esta seja modificada. Desta forma, para calcular o consumo de potência de um determinado tráfego, depois de codificado por um destes cinco esquemas, basta utilizar os macromodelos dos módulos da NoC original e dos módulos de codificação e decodificação do esquema adotado.

Tabela 4.6: Macromodelos lineares dos módulos de diferentes esquemas de codificação com largura de *flit* igual a 8 bits.

Módulo	P_0	R
Encoder (*APE*) – *Gray*	1,76	2,27
Decoder (*APD*) – *Gray*	1,51	4,36
Encoder (*APE*) – *Adaptive Prob. Encoding*	12,1	3,62
Decoder (*APD*) – *Adaptive Prob. Encoding*	9,78	3,79
Encoder (*APE*) – *Transition*	2,35	2,89
Decoder (*APD*) – *Transition*	1,96	3,2
Encoder (*APE*) – *T-Bus-Invert*	3,18	6,16
Decoder (*APD*) – *T-Bus-Invert*	2,1	1,54

Como já comentado anteriormente, a utilização do esquema de codificação *Bus-Invert* necessita que a NoC seja alterada, inserindo bits de controle, o que aumenta a largura de *flit* e, consequentemente, o consumo de potência no buffer, na lógica de controle e nos canais de comunicação. Por este motivo, para calcular o consumo de potência com tráfego codificado através deste esquema, deve-se utilizar os macromodelos da NoC alterada, juntamente com os macromodelos dos módulos do esquema de codificação *Bus-Invert*. A Tabela 4.7 mostra estes macromodelos, considerando que o tráfego codificado possui largura de *flit* igual a 9 bits.

Tabela 4.7: Macromodelos lineares dos módulos de uma NoC Hermes adaptada ao esquema *Bus-Invert*, com largura de *flit* igual a 9 bits e buffers de 16 palavras, e macromodelos dos módulos deste esquema de codificação para a mesma largura de *flit*.

Módulo	P_0	R
Encoder (*APE*) – *Bus-Invert*	1,17	2,95
Decoder (*APD*) – *Bus-Invert*	0,55	0,25
Buffer (APB_F)	11,49	22,13
Control (APS_C)	4,39	0,98
Canal entre roteadores (APL_R)	0,19	0,8

A Tabela 4.8 apresenta os macromodelos lineares dos módulos de uma NoC Hermes com largura de *flit* igual a 16 bits. A Tabela 4.9 mostra os macromodelos dos módulos dos esquemas de codificação *Gray*, *Transition* e *T-Bus-Invert*, implementados para esta NoC, sem necessidade de alteração da mesma. Já a Tabela 4.10 apresenta os macromodelos dos módulos do esquema *Bus-Invert* com 1 *cluster* e dos módulos da NoC adaptada a este esquema, com largura de *flit* igual a 17 bits e utilizando buffers de 16 palavras. Similarmente, a Tabela 4.11 apresenta os macromodelos lineares dos módulos de uma NoC Hermes adaptada ao esquema *Bus-Invert* com 2 *clusters*, com largura de *flit* igual a 18 bits e buffers de 16 palavras, e macromodelos dos módulos deste esquema de codificação para a mesma largura de *flit*.

Tabela 4.8: Macromodelos lineares dos módulos de uma NoC Hermes com largura de *flit* igual a 16 bits.

Módulo	P_0	R
Buffer (APB_F) – 4 palavras	6,5	8,8
Buffer (APB_F) – 8 palavras	10,72	19,48
Buffer (APB_F) – 16 palavras	17,82	38,93
Control (APS_C)	4,4	1,36
Canal entre roteadores (APL_R)	0,2	1,42

Tabela 4.9: Macromodelos lineares dos módulos de diferentes esquemas de codificação com largura de flit igual a 16 bits.

Módulo	P_0	R
Encoder (*APE*) – *Gray*	2,9	4,64
Decoder (*APD*) – *Gray*	2,43	15,8
Encoder (*APE*) – *Transition*	4,06	5,85
Decoder (*APD*) – *Transition*	3,33	6,47
Encoder (*APE*) – *T-Bus-Invert*	6,26	12,36
Decoder (*APD*) – *T-Bus-Invert*	4,09	2,92

Tabela 4.10: Macromodelos lineares dos módulos de uma NoC Hermes adaptada ao esquema Bus-Invert com 1 cluster, com largura de flit igual a 17 bits e buffers de 16 palavras, e macromodelos dos módulos deste esquema de codificação para a mesma largura de flit.

Módulo	P_0	R
Encoder (*APE*) – *Bus-Invert*	2,35	8,19
Decoder (*APD*) – *Bus-Invert*	1,10	1,21
Buffer (APB_F)	18,77	42,73
Control (APS_C)	4,42	1,98
Canal entre roteadores (APL_R)	0,2	1,51

Tabela 4.11: Macromodelos lineares dos módulos de uma NoC Hermes adaptada ao esquema Bus-Invert com 2 clusters, com largura de flit igual a 18 bits e buffers de 16 palavras, e macromodelos dos módulos deste esquema de codificação para a mesma largura de flit.

Módulo	P_0	R
Encoder (*APE*) – *Bus-Invert*	2,42	9,05
Decoder (*APD*) – *Bus-Invert*	1,10	1,21
Buffer (APB_F)	19,62	44,58
Control (APS_C)	4,42	1,98
Canal entre roteadores (APL_R)	0,21	1,59

A Tabela 4.12 apresenta os macromodelos lineares dos módulos de uma NoC Hermes com largura de *flit* igual a 32 bits. A Tabela 4.13 mostra os macromodelos dos

módulos dos esquemas de codificação *Gray*, *Transition* e *T-Bus-Invert*, implementados para esta NoC, sem necessidade de alteração da mesma. Por fim, a Tabela 4.14 apresenta os macromodelos dos módulos do esquema *Bus-Invert* com 4 *clusters* e dos módulos da NoC adaptada a este esquema, com largura de *flit* igual a 32 bits e utilizando buffers de 16 palavras.

Tabela 4.12: Macromodelos lineares dos módulos de uma NoC Hermes com largura de flit igual a 32 bits.

Módulo	P_0	R
Buffer (APB_F) – 4 palavras	11,92	26,3
Buffer (APB_F) – 8 palavras	19,66	46,34
Buffer (APB_F) – 16 palavras	32,69	78,56
Control (APS_C)	4,41	3,56
Canal entre roteadores (APL_R)	0,23	2,83

Tabela 4.13: Macromodelos lineares dos módulos de diferentes esquemas de codificação com largura de flit igual a 32 bits.

Módulo	P_0	R
Encoder (*APE*) – *Gray*	5,15	9,4
Decoder (*APD*) – *Gray*	4,21	59,24
Encoder (*APE*) – *Transition*	7,48	11,77
Decoder (*APD*) – *Transition*	6,02	13,03
Encoder (*APE*) – *T-Bus-Invert*	12,46	24,82
Decoder (*APD*) – *T-Bus-Invert*	8,09	5,84

Através destes macromodelos, pode-se então calcular os parâmetros APB_F, APS_C, *APE*, *APD* e APL_R e obter a estimativa de consumo de potência na comunicação entre dois núcleos da NoC, utilizando-se a Equação (8), descrita anteriormente na seção 4.2. Da mesma forma pode-se calcular estes mesmos parâmetros para obter a estimativa de consumo de potência na comunicação entre dois núcleos da NoC, utilizando-se um esquema de codificação, através da Equação (9). O próximo Capítulo apresenta a análise do consumo de potência, com base nestas equações.

Tabela 4.14: Macromodelos lineares dos módulos de uma NoC Hermes adaptada ao esquema Bus-Invert com 4 clusters, com largura de flit igual a 36 bits e buffers de 16 palavras, e macromodelos dos módulos deste esquema de codificação para a mesma largura de flit.

Módulo	P_0	R
Encoder (*APE*) – *Bus-Invert*	4,68	11,78
Decoder (*APD*) – *Bus-Invert*	2,19	0,99
Buffer (APB_F)	36,15	90,19
Control (APS_C)	4,42	4,03
Canal entre roteadores (APL_R)	0,23	3,19

4.6 Conclusão

Este capítulo apresentou o modelo de consumo de potência em NoCs, proposto no escopo deste trabalho, explicando como foi feita a definição deste modelo, bem o fluxo utilizado para a obtenção dos parâmetros de potência. Também neste capítulo foram apresentados os resultados de área dos módulo da NoC e dos módulos dos esquemas de codificação, além dos macromodelos de potência para cada um destes módulos. Os macromodelos são criados a partir da observação de que o consumo de potência aumenta linearmente com o aumento das transições. Desta forma, é possível utilizá-los para estimar o consumo de cada módulo de acordo com a atividade de transição em seus sinais de entrada, como será apresentado no capitulo a seguir.

5 ANÁLISE DO CONSUMO DE POTÊNCIA

Como mostrado no capítulo anterior, a estimativa da potência média consumida na comunicação entre os núcleos de um System-on-Chip depende da infra-estrutura de interconexão (aqui assumida como sendo a rede Hermes) e do tráfego gerado por estes núcleos. Este capítulo apresenta a forma como é calculada a potência média consumida na NoC nas duas abordagens, com e sem esquemas de codificação. A análise da potência média consumida nos módulos foi feita com diferentes padrões de tráfego reais inseridos em pacotes e transmitidos através da NoC.

A fim de avaliar completamente o efeito do tráfego resultante de uma aplicação real, os experimentos devem ser executados com quantidades realistas de dados, com até milhares de pacotes. Entretanto, o tempo de simulação para estas quantidades de pacotes em um simulador SPICE é impraticável. Sendo assim, uma alternativa foi tomada, explorando a possibilidade de inserir os macromodelos em um modelo com mais alto nível de abstração, o qual foi simulado com o ambiente *Ptolemy II* (BROOKS et al., 2005). Este, por sua vez, inclui um modelo do codificador, dos componentes da NoC e do decodificador. Estes modelos são utilizados para calcular a percentagem de transições de bit em cada padrão de tráfego e, com base nos macromodelos de potência dos módulos, estima-se a potência média consumida na comunicação, de acordo com o tráfego.

A Figura 5.1 mostra a interface do ambiente *Ptolemy II*. A lista com pastas à esquerda na Figura, mostra os componentes primitivos da ferramenta, os quais podem ser utilizados para criar novos componentes. Na área maior, à direita, é onde o usuário insere componentes e cria conexões entre os mesmos para descrever o sistema a ser simulado. No exemplo mostrado na Figura 5.1 é possível observar um grupo de componentes responsáveis por gerar um tráfego de entrada para o codificador. O tráfego é lido de um arquivo-texto de entrada e transmitido bit a bit nos *n* fios do canal de comunicação, onde *n* corresponde à largura deste canal. O codificador recebe o tráfego de entrada e executa sua função de codificação, gerando um novo tráfego em sua saída. Os analisadores recebem ambos os tráfegos, original e codificado, e verificam a percentagem de transições em ambos os casos, calculando o consumo de potência com cada padrão de tráfego.

O codificador possui uma estrutura hierárquica, onde cada caixa cinza corresponde a um nível interno da hierarquia, como se pode observar na Figura 5.2, que mostra um zoom com a estrutura interna do codificador *Bus-Invert*. Alguns destes codificadores apresentam uma estrutura interna bastante complexa, devido à sua máquina-de-estados. É o caso do codificador *T-Bus-Invert*, ilustrado na Figura 5.3.

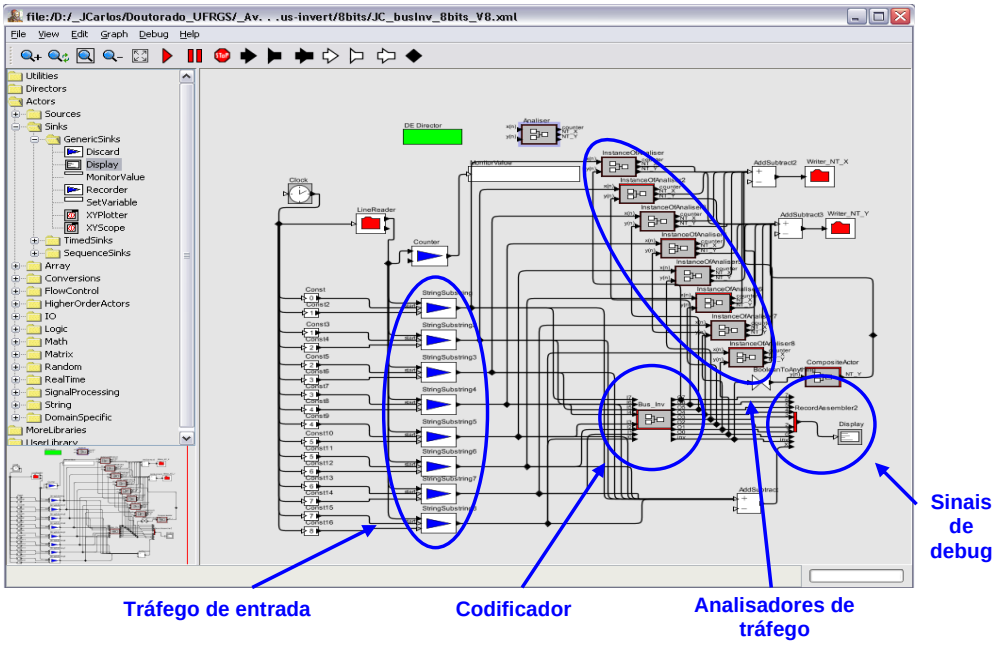


Figura 5.1: Interface do ambiente *Ptolemy II*.

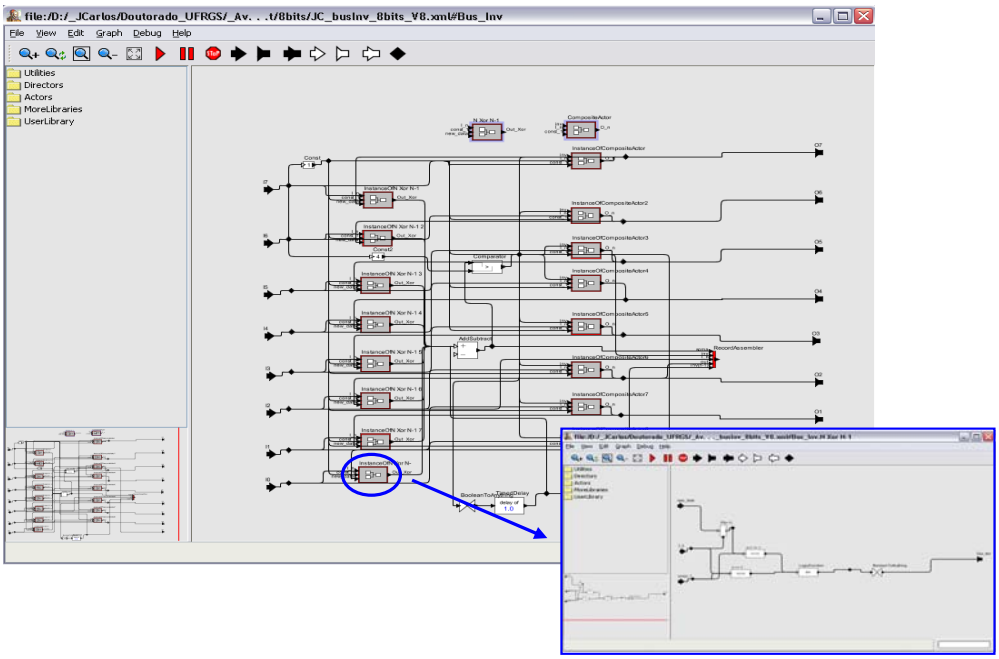


Figura 5.2: Estrutura interna do codificador *Bus-Invert*.

A estimativa de consumo de potência é feita da maneira descrita a seguir. Consideremos, por exemplo, um padrão de tráfego com 2000 *flits* de 8 bits. A quantidade máxima de transições nestes 2000 *flits* seria 15992, com o número máximo de transições (1999) nos 8 bits do *flit* (1999 * 8 = 15992 = 100% de transições). Considerando agora, por exemplo, uma simulação onde obteviram-se 11994 transições. Neste caso, a quantidade de transições corresponde a 80% do máximo possível (11994 = 80% de 15992). Assim, a potência média consumida no buffer, na lógica de controle,

e no canal de comunicação pode ser calculada com base na taxa de 80% de transições, como mostrado a seguir:

$$APB_F = P_0 + \%T * R$$
$$APB_F = 10,61 + 0,8 * 19,19$$
$$APB_F = \underline{25,96\ mW}$$

$$APS_C = P_0 + \%T * R$$
$$APS_C = 4,31 + 0,8 * 0,8$$
$$APS_C = \underline{4,95\ mW}$$

$$APL_R = P_0 + \%T * R$$
$$APL_R = 0,19 + 0,8 * 0,71$$
$$APL_R = \underline{0,76\ mW}$$

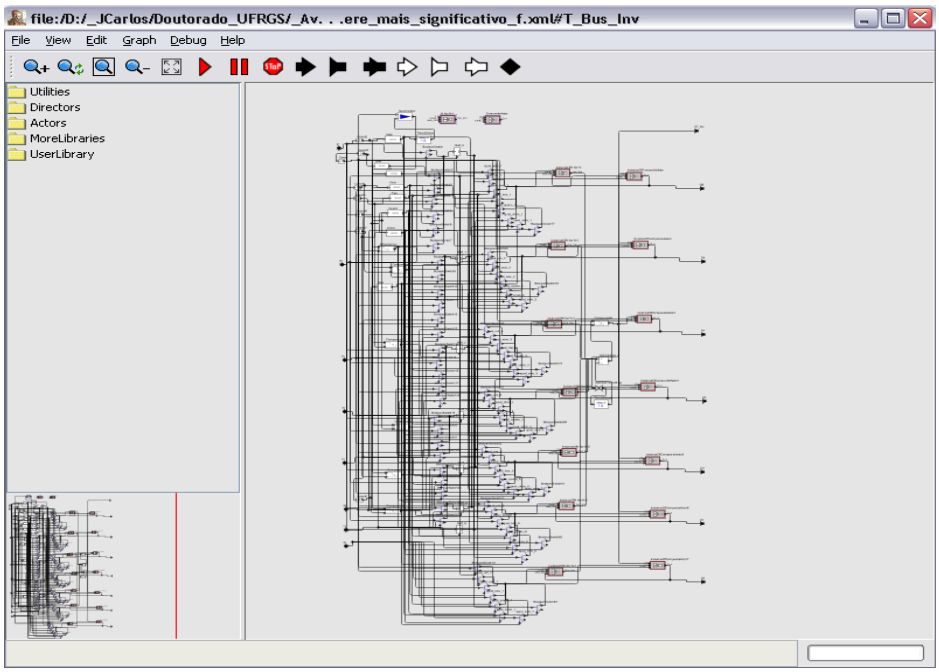


Figura 5.3: Estrutura interna do codificador *T-Bus-Invert*.

A Figura 5.4 ilustra a localização do valor estimado para o consumo de potência em um buffer Hermes de 16 palavras, com largura de *flit* igual a 8 bits, e com 80% de transição de sinais no tráfego recebido. Utilizando-se a Equação (8) é possível calcular a potência média consumida por um pacote em um *hop* como mostrado a seguir.

$$RRPij = APB_F + APS_C + APL_R$$
$$RRPij = 25,96 + 4,95 + 0,76 = \underline{31,67\ mW}$$

Supondo agora que, em uma NoC que utiliza o esquema de codificação *Adaptive Encoding*, a quantidade de transições de sinais seja reduzida dos 11994 do tráfego

original para 3998 (30% do máximo possível, ou seja 30% de 15992). Neste caso, a potência média consumida em um *hop* será:

$APB_F = 10,61 + 0,3 * 19,19$
$APB_F = \underline{16,36\ mW}$

$APS_C = 4,31 + 0,3 * 0,8$
$APS_C = \underline{4,55\ mW}$

$APL_R = 0,19 + 0,3 * 0,71$
$APL_R = \underline{0,4\ mW}$

$RRPij = 16,36 + 4,55 + 0,4 = \underline{21,31\ mW}$

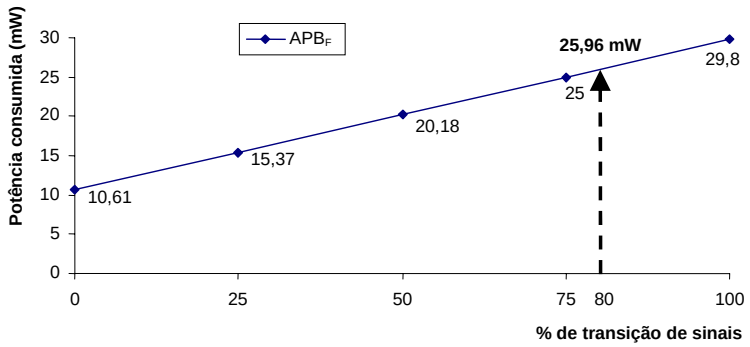


Figura 5.4: Consumo de potência estimada para um buffer Hermes de 16 palavras, com largura de *flit* igual a 8 bits, e com 80% de transição de sinais no tráfego recebido.

Também é necessário que se inclua neste cálculo a potência média consumida nos módulos *encoder* e *decoder*. O consumo de potência no encoder deve ser calculada com base na percentagem de chaveamento do tráfego original, que é recebido nos sinais de entrada deste módulo. Neste exemplo, o tráfego original tinha 80% da quantidade máxima possível de transições se sinais. Neste caso, a potência média consumida no módulo *encoder* será:

$APE = 12,1 + 0,8 * 3,62$
$APE = \underline{14,99\ mW}$

Entretanto, a potência média consumida no módulo *decoder* deve ser calculada com base na taxa de transições do tráfego codificado, recebido nos sinais de entrada deste módulo, ou seja, 30%.

$APD = 9,78 + 0,3 * 3,79$
$APD = \underline{10,91\ mW}$

A redução na potência consumida por *hop*, neste exemplo, foi de 10,36 mW (31,67 mW com o tráfego original – 21,31 mW com o tráfego codificado). Por outro lado, o esquema de codificação adiciona um consumo médio de potência de 25,9 mW (14,99 mW do *encoder* + 10,91 mW do *decoder*). Considerando uma redução de pelo menos 10,36 mW em cada *hop*, pode-se assumir que após 3 *hops* a potência consumida pelo esquema de codificação pode ser amortizada. Em 3 *hops* já haverá uma redução de 31,08 mW nos roteadores, contra um consumo extra de 25,9 mW das etapas de codificação e decodificação, que ocorrem somente uma vez em cada transmissão.

Vamos considerar agora este mesmo tráfego em uma NoC utilizando o esquema de codificação *Bus-Invert*. Como visto no capítulo anterior, este esquema necessita de um bit a mais nos buffers, na lógica de controle e nos canais de comunicação. Também foi visto que o consumo de potência aumenta um pouco para estes componentes, com a inclusão de 1 bit no *flit* da NoC. Sendo assim, o consumo de potência nos buffers, na lógica de controle e no canal de comunicação deve ser re-calculado com base no macromodelo da NoC com o esquema *Bus-Invert*.

Supondo, então, que este esquema de codificação também reduza a quantidade de transições para 30%, a potência média consumida em um *hop* será:

$APB_F = 11,49 + 0,3 * 22,13$
$APB_F = \underline{18,13\ mW}$

$APS_C = 4,39 + 0,3 * 0,98$
$APS_C = \underline{4,68\ mW}$

$APL_R = 0,19 + 0,3 * 0,8$
$APL_R = \underline{0,43\ mW}$

$RRPij = 18,13 + 4,68 + 0,43 = \underline{23,24\ mW}$

O próximo passo é cálcular a potência média consumida nos módulos *encoder* e *decoder*. Da mesma forma que no esquema anterior, a potência média consumida no módulo *encoder* deve ser calculada com base no tráfego original, com a taxa de transição de sinais igual a 80%, enquanto que a potência do *decoder* deve ser calculada com base na taxa de transições do tráfego codificado (30%), como segue:

$APE = 1,17 + 0,8 * 2,95$
$APE = \underline{3,53\ mW}$

$APD = 0,55 + 0,3 * 0,25$
$APD = \underline{0,62\ mW}$

Nesta nova abordagem, a redução na potência média consumida por *hop* foi de 8,43 mW (31,67 mW – 23,24 mW). Isto diz respeito à potência consumida no roteador original, sob efeito do tráfego original, comparada com a potência consumida no

roteador adaptado para o esquema *Bus-Invert*, sob efeito do tráfego codificado. Pode-se, então, considerar uma redução de pelo menos 8,43 mW a cada *hop*.

Em contrapartida, o esquema de codificação adiciona um consumo médio de potência de 4,15 mW (3,53 mW + 0,62 mW). Desta forma, pode-se assumir que, em apenas 1 *hop*, a potência consumida pelo esquema de codificação já pode ser amortizada, pois neste primeiro hop já haverá uma redução de 8,43 mW contra um consumo extra de codificação de 4,15 mW.

5.1 Resultados Experimentais

Esta seção apresenta os resultados experimentais obtidos através de simulações em nível de sistema executadas no ambiente *Ptolemy II*, utilizando os macromodelos descritos na seção 4.5 para diferentes padrões de tráfego reais. Os tráfegos simulados foram: "html" (página com componentes da web), "gzip" (arquivo compactado), "gcc"(executável), "bytecode"(classe java), "wav" e "mp3" (áudio), "raw", "bmp", "jpg" e "tiff" (imagens) e "pdf" (arquivo de texto com imagens).

As tabelas apresentadas nesta seção mostram, na primeira coluna, o tipo de tráfego utilizado. A segunda coluna mostra a redução na atividade de transição de sinais obtida nos experimentos executados no escopo deste trabalho, reproduzindo cada esquema de codificação. Esta redução corresponde à quantidade de transições do tráfego codificado quando comparado ao tráfego original. A terceira coluna mostra a média de potência consumida (*APH*, ou seja, APB_F + APS_C + APL_R) sem o uso de técnicas de codificação (tráfego original). A quarta coluna mostra a mesma medida (*APH*) com o uso do esquema de codificação. A quinta coluna mostra o consumo extra de potência devido à inserção dos módulos de codificação e decodificação no sistema (*APE* + *APD*). A sexta e última coluna diz respeito à quantidade de *hops* necessários para amortizar o custo extra da codificação.

A Tabela 5.1 apresenta os resultados obtidos com o esquema *Adaptive Probability Encoding* em uma NoC Hermes com largura de *flit* igual a 8 bits. Em praticamente metade dos tráfegos simulados este esquema aumentou a atividade de transição de sinais, comparando-se o tráfego codificado com o tráfego original, aumentando também o consumo de potência na NoC. Nestes casos, marcados com traços na última coluna da tabela, não é possível amortizar o custo da codificação. Em outros casos, mesmo reduzindo a atividade de transição de sinais, são necessários muitos *hops* para amortizar o custo de potência dos módulos de codificação e decodificação, devido ao consumo elevado de potência destes módulos, como se pode obervar na quinta coluna desta tabela. O melhor resultado obtido com o esquema *Adaptive Probability Encoding* foi com o tráfego "wav", onde a redução da atividade de transição foi de 21,95% e o custo da codificação é amortizado após 11 *hops* (PALMA et al., 2007-a).

A Tabela 5.2 apresenta os resultados obtidos com o esquema *Bus-Invert* em uma NoC Hermes com largura de *flit* igual a 8 bits. A terceira coluna, é calculada com base no macromodelo da NoC original, sem esquemas de codificação, enquanto que a quarta coluna é calculada com base no macromodelo utilizando *Bus-Invert* (com 1 bit a mais em todos os módulos). Os resultados apresentados nesta tabela mostram que o consumo de potência na NoC aumentou com o uso deste esquema de codificação, apesar do fato de que o mesmo reduziu a transição de sinais com todos os tráfegos simulados. Isto acontece devido à inclusão do bit extra de controle em todos os componentes da NoC, aumentando seu consumo de potência (PALMA et al., 2007-a).

Tabela 5.1: Resultados obtidos com o esquema *Adaptive Probability Encoding* em uma NoC Hermes com largura de *flit* igual a 8 bits.

Stream	Redução na transição de sinais	Pôtência média consumida na NoC sem codificação (APB_F+APS_C+APL_R)	Pôtência média consumida na NoC com codificação (APB_F + APS_C + APL_R)	Pôtência média consumida pelos módulos de codificação (*APE* + *APD*)	# of hops
HTML	-1,4 %	22,24 mW	22,34 mW ☹	24,1 mW	-
GZIP	1,03%	25,5 mW	25,4 mW	25,6 mW	240
GCC	0,91 %	24,69 mW	24,6 mW	25,27 mW	292
Bytecode	9,3 %	23,45 mW	22,68 mW	24,7 mW	32
WAV	21,95 %	25,5 mW	23,25 mW	25,17 mW	11
MP3	- 2,41 %	24,8 mW	25 mW ☹	25,38 mW	-
RAW	- 10,98 %	22,24 mW	23 mW ☹	24,5 mW	-
BMP	- 0,5 %	25,3 mW	25,36 mW ☹	25,5 mW	-
JPG	0,8 %	25,46 mW	25,38 mW	25,5 mW	327
TIFF	-1,16 %	25,34 mW	25,46 mW ☹	25,55 mW	-
PDF	6,61 %	26,06 mW	25,34 mW	25,65 mW	36

A Figura 5.5 mostra o consumo de potência em um buffer Hermes de 16 palavras, com larguras de *flit* igual a 8 bits e 9 bits. Como se pode observar no caso apresentado nesta figura, mesmo reduzindo a percentagem de transição de sinais, o consumo de potência com tráfego codificado foi maior do que com o tráfego original, transmitido na NoC original. Isto aconteceu com quase todos os tráfegos aprentados na Tabela 5.2, exceto com o tráfego "pdf". Neste caso a redução foi bastante significativa (23,15%), amortizando o custo da codificação após 13 *hops*.

Tabela 5.2: Resultados obtidos com o esquema *Bus-Invert* em uma NoC Hermes com largura de *flit* igual a 8 bits.

Stream	Redução na transição de sinais	Pôtência média consumida na NoC sem codificação (APB_F + APS_C + APL_R)	Pôtência média consumida na NoC com codificação (APB_F + APS_C + APL_R)	Pôtência média consumida pelos módulos de codificação (*APE* + *APD*)	# of hops
HTML	6,2 %	21,33 mW	22,75 mW ☹	2,93 mW	-
GZIP	18,7%	25,5 mW	25,79 mW ☹	3,75 mW	-
GCC	17,9%	18,46 mW	19,18 mW ☹	2,36 mW	-
Bytecode	12 %	23,45 mW	24,49 mW ☹	3,35 mW	-
WAV	18,8 %	25,52 mW	25,8 mW ☹	3,75 mW	-
MP3	18,48 %	25,3 mW	25,63 mW ☹	3,7 mW	-
RAW	14,6 %	22,24 mW	23 mW ☹	3,1 mW	-
BMP	18,2%	25,3 mW	25,66 mW ☹	3,71 mW	-
JPG	19,5 %	25,46 mW	25,65 mW ☹	3,74 mW	-
TIFF	18,3 %	25,25 mW	25,6 mW ☹	3,7 mW	-
PDF	23,15 %	26 mW	25,76 mW	3,85 mW	13

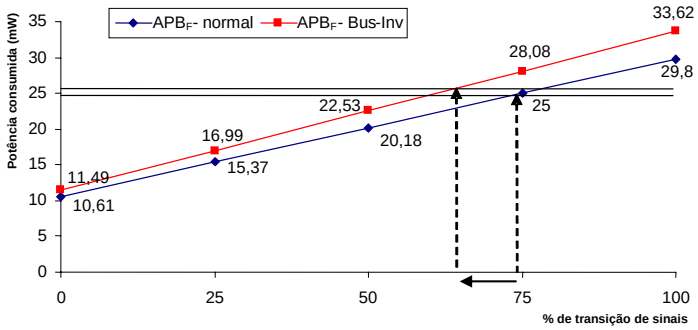


Figura 5.5: Consumo de potência em um buffer Hermes de 16 palavras, com larguras de *flit* igual a 8 bits e 9 bits. Mesmo reduzindo a percentagem de transição de sinais, o consumo de potência com tráfego codificado foi maior do que com o tráfego original, transmitido na NoC original.

A Tabela 5.3 mostra os resultados obtidos com o esquema *Gray* em uma NoC com largura de flit igual a 8 bits. Na maioria dos tráfegos simulados este esquema aumentou a percentagem de transição de sinais, aumentando também o consumo de potência na NoC. O melhor caso é com o tráfego "bmp", onde o custo da codificação é amortizado após 7 hops, devido a uma redução de 11,82%. Em outros casos, a amortização acontece após 18 e 19 hops. De forma similar ao esquema *Gray*, o esquema *Transition* também aumentou a percentagem de transição de sinais na maioria dos tráfegos simulados, como mostra a Tabela 5.4. Porém, os resultados com este esquema foram um pouco melhores do que o anterior, apresentando, nos melhores casos, 6 e 9 *hops* para a amortização, com reduções de 12,22% e 9,22% (PALMA et al., 2007-b).

Tabela 5.3: Resultados obtidos com o esquema *Gray* em uma NoC Hermes com largura de *flit* igual a 8 bits.

Stream	Redução na transição de sinais	Pôtência média consumida na NoC sem codificação ($APB_F + APS_C + APL_R$)	Pôtência média consumida na NoC com codificação ($APB_F + APS_C + APL_R$)	Pôtência média consumida pelos módulos de codificação ($APE + APD$)	# of hops
HTML	- 11,36 %	22,24 mW	23,04 mW ⊗	5,71 mW	-
GZIP	- 0,49 %	25,5 mW	25,55 mW ⊗	6,6 mW	-
GCC	- 5,57 %	18,47 mW	18,65 mW ⊗	4,36 mW	-
Bytecode	0,21 %	23,56 mW	23,54 mW	5,96 mW	331
WAV	- 0,01 %	25,51 mW	25,51 mW ⊗	6,59 mW	-
MP3	- 0,06 %	25,31 mW	25,32 mW ⊗	6,53 mW	-
RAW	4,11 %	22,24 mW	21,95 mW	5,48 mW	19
BMP	11,82 %	21,22 mW	20,5 mW	5,06 mW	7
JPG	3,42 %	24,55 mW	24,23 mW	6,22 mW	19
TIFF	- 1,36 %	24,65 mW	24,78 mW ⊗	6,34 mW	-
PDF	3,5 %	26,06 mW	25,68 mW	6,69 mW	18

Os melhores resultados para uma NoC com largura de *flit* igual a 8 bits foram obtidos com o esquema *T-Bus-Invert*, proposto neste trabalho. Estes resultados são apresentados na Tabela 5.5. Como mostrado na quinta coluna desta tabela, o custo da codificação para este esquema é um pouco mais elevado do que nos esquemas *Gray* e

Transition. Por outro lado, a redução da atividade de transição de sinais é bem maior, chegando a 30,57% no melhor caso. Na maioria dos casos, a quantidade de *hops* necessários para amortizar o custo da codificação foi de apenas 3 ou 4. Este esquema utiliza a vantagem da ampla redução na transição de sinais proporcionada pela técnica do esquema *Bus-Invert*, porém sem inserir bits extras nos módulos e canais de comunicação da NoC (PALMA et al., 2007-b).

Tabela 5.4: Resultados obtidos com o esquema Transition em uma NoC Hermes com largura de *flit* igual a 8 bits.

Stream	Redução na transição de sinais	Pôtência média consumida na NoC sem codificação ($APB_F + APS_C + APL_R$)	Pôtência média consumida na NoC com codificação ($APB_F + APS_C + APL_R$)	Pôtência média consumida pelos módulos de codificação ($APE + APD$)	# of hops
HTML	- 2,97 %	22,24 mW	22,45 mW ⊗	6,42 mW	-
GZIP	1,17 %	25,5 mW	25,38 mW	7,34 mW	61
GCC	2,38 %	24,68 mW	24,46 mW	7,07 mW	31
Bytecode	9,22 %	23,18 mW	22,44 mW	6,55 mW	9
WAV	12,22 %	24,45 mW	23,32 mW	6,87 mW	6
MP3	- 0,21 %	25,3 mW	25,33 mW ⊗	7,3 mW	-
RAW	- 12,44 %	22,24 mW	23,12 mW ⊗	6,53 mW	-
BMP	- 0,02 %	25,3 mW	25,3 mW ⊗	7,3 mW	-
JPG	- 0,6 %	24,55 mW	24,61 mW ⊗	7,08 mW	-
TIFF	- 1,16 %	25,25 mW	25,37 mW ⊗	7,3 mW	-
PDF	7,52 %	26,06 mW	25,24 mW	7,39 mW	9

Tabela 5.5: Resultados obtidos com o esquema *T-Bus-Invert* em uma NoC Hermes com largura de *flit* igual a 8 bits.

Stream	Redução na transição de sinais	Pôtência média consumida na NoC sem codificação ($APB_F + APS_C + APL_R$)	Pôtência média consumida na NoC com codificação ($APB_F + APS_C + APL_R$)	Pôtência média consumida pelos módulos de codificação ($APE + APD$)	# of hops
HTML	9,8 %	22,24 mW	21,55 mW	8,27 mW	12
GZIP	26,89 %	25,5 mW	22,73 mW	9,45 mW	3
GCC	26,35 %	24,68 mW	22,18 mW	9,11 mW	4
Bytecode	20,88 %	23,56 mW	21,81 mW	8,72 mW	5
WAV	28,19 %	24,45 mW	21,84 mW	8,99 mW	3
MP3	27,09 %	25,3 mW	22,57 mW	9,36 mW	3
RAW	14,5 %	22,24 mW	21,22 mW	8,22 mW	8
BMP	26,38 %	25,3 mW	22,63 mW	9,37 mW	4
JPG	26,26 %	25,06 mW	22,47 mW	9,28 mW	4
TIFF	27,68 %	25,26 mW	22,47 mW	9.34 mW	3
PDF	30,57 %	26,06 mW	22,74 mW	9,62 mW	3

Como mencionado na seção 4.4, o esquema *Adaptive Probability Encoding* não foi implementado na NoC Hermes com largura de *flit* igual a 16 bits. No lugar deste esquema implementou-se o esquema *Bus-Invert* com 2 *clusters*, cujos resultados são apresentados na Tabela 5.6. A segunda coluna desta tabela mostra que a redução da atividade de transição de sinais foi por volta de 23% na maioria dos casos, um resultado

um pouco melhor do que na NoC com largura de *flit* igual a 8 bits, onde esta redução foi por volta de 18% na maioria dos casos. Considerando que esta abordagem com 2 *clusters* de 8 bits funciona como 2 codificadores *Bus-Invert* de 8 bits em paralelo, esta diferença pode ser causada pela relação entre *flits* consecutivos enviados através da NoC. Quando se utiliza canais de 16 bits, a atividade de transição de sinais é computada entre os bytes de mesma posição em *flits* consecutivos (enviando 2 bytes em cada *flit*). Quando se utiliza canais de 8 bits, a atividade de transição de sinais é computada entre os bytes consecutivos do dado transmitido.

Na implementação deste esquema, utilizando 2 *clusters* em uma NoC com *flits* de 16 bits, o consumo de potência foi reduzido na maioria dos tráfegos simulados. No melhor caso, com o tráfego "wav", a redução da atividade de transição foi de 29,04% e o custo da codificação foi amortizado após 6 *hops* (PALMA et al., 2007-b).

Tabela 5.6: Resultados obtidos com o esquema Bus-Invert com 2 clusters em uma NoC Hermes com largura de *flit* igual a 16 bits.

Stream	Redução na transição de sinais	Pôtência média consumida na NoC sem codificação ($APB_F + APS_C + APL_R$)	Pôtência média consumida na NoC com codificação ($APB_F + APS_C + APL_R$)	Pôtência média consumida pelos módulos de codificação ($APE + APD$)	# of hops
HTML	11,73 %	37,16 mW	39,17 mW ⊗	7 mW	-
GZIP	22,83 %	43,17 mW	42,67 mW	8,47 mW	17
GCC	20 %	41,26 mW	41,57 mW ⊗	8,02 mW	-
Bytecode	17,08 %	37,9 mW	38,98 mW ⊗	7,23 mW	-
WAV	29,04 %	38,93 mW	37,7 mW	7,42 mW	6
MP3	23,14 %	43,02 mW	42,47 mW	8,43 mW	15
RAW	21,41 %	38,51 mW	38,77 mW ⊗	7,36 mW	-
BMP	22,81 %	42,94 mW	42,47 mW	8,41 mW	18
JPG	23,27 %	42,78 mW	42,22 mW	8,37 mW	15
TIFF	23,33 %	42,62 mW	42,06 mW	8,34 mW	15
PDF	23,99 %	42,82 mW	42,09 mW	8,38 mW	11

A Tabela 5.7 mostra os resultados obtidos com o esquema *Bus-Invert* implementado com somente 1 *cluster* em uma NoC com *flits* de 16 bits. Conforme descrito na segunda coluna desta tabela, a redução da atividade de transição de sinais foi menor do que na implementação com 2 *clusters*. Isto comprova a afirmação dos autores do *Bus-Invert*, em (STAN; BURLESON, 1995), indicando que a utilização de *clusters* de 8 bits é a mais eficiente na redução das transições. Entretando, em uma NoC com *flits* de 16 bits, a implementação com 1 *cluster* de 16 bits foi mais eficiente na redução do consumo de potência. Isto porque o impacto da inserção de 2 bits extras nos componentes da NoC (no caso implementação com 2 *clusters*) é significante e não é compensado por um pequeno aumento na redução das transições, comparando-a com a implementação de 1 cluster. Conforme a Tabela 5.7, na maioria dos casos o custo da codificação é amortizado entre 4 e 6 *hops*, onde a redução da atividade de transição chega a 22,69% (PALMA et al., 2007-b).

Tabela 5.7: Resultados obtidos com o esquema *Bus-Invert* com 1 *cluster* em uma NoC Hermes com largura de *flit* igual a 16 bits.

Stream	Redução na transição de sinais	Pôtência média consumida na NoC sem codificação ($APB_F + APS_C + APL_R$)	Pôtência média consumida na NoC com codificação ($APB_F + APS_C + APL_R$)	Pôtência média consumida pelos módulos de codificação ($APE + APD$)	# of hops
HTML	10,57 %	37,16 mW	37,9 mW ⊗	6,6 mW	-
GZIP	19,59 %	43,17 mW	41,81 mW	7,84 mW	6
GCC	16,86 %	41,26 mW	40,67 mW	7,45 mW	13
Bytecode	12,89 %	37,9 mW	38,25 mW ⊗	6,75 mW	-
WAV	22,69 %	38,93 mW	37,46 mW	6,92 mW	5
MP3	19,9 %	43,02 mW	41,61 mW	7,81 mW	6
RAW	17,2 %	38,51 mW	38,08 mW	6,86 mW	16
BMP	19,19 %	42,94 mW	41,7 mW	7,8 mW	6
JPG	19,99 %	42,78 mW	41,38 mW	7,76 mW	6
TIFF	20,75 %	42,62 mW	41,06 mW	7,72 mW	5
PDF	21,52 %	42,82 mW	41,07 mW	7,76 mW	4

A Tabela 5.8 mostra os resultados da implementação do esquema *Gray* em uma NoC Hermes com largura de *flit* igual a 16 bits, enquanto que e a Tabela 5.9 apresenta os resultados do esquema *Transition* nesta mesma configuração da NoC. Assim como na implementação com *flits* de 8 bits, estes esquemas aumentaram a atividade de transição de sinais para praticamente metade dos tráfegos simulados. Em outros casos, a redução foi pequena, necessitando de um número elevado de *hops* para amortizar o custo da codificação. O esquema *Gray* apresentou como melhor resultado 10 *hops* para o tráfego "wav", com redução de 7,91% da atividade de transição. Já o esquema *Transition* apresentou como bons resultados 4 *hops* utilizando o tráfego "wav", com 19,67% de redução nas transições e 6 *hops* com o tráfego "gcc", reduzindo em 11,78% a atividade de transição de sinais (PALMA et al., 2007-b).

Tabela 5.8: Resultados obtidos com o esquema Gray em uma NoC Hermes com largura de *flit* igual a 16 bits.

Stream	Redução na transição de sinais	Pôtência média consumida na NoC sem codificação ($APB_F + APS_C + APL_R$)	Pôtência média consumida na NoC com codificação ($APB_F + APS_C + APL_R$)	Pôtência média consumida pelos módulos de codificação ($APE + APD$)	# of hops
HTML	- 32,07 %	37,16 mW	41,84 mW ⊗	14,29 mW	-
GZIP	- 0,45 %	43,16 mW	43,25 mW ⊗	15,5 mW	-
GCC	- 4,09 %	29,7 mW	29,99 mW ⊗	8,96 mW	-
Bytecode	- 13,42 %	37,9 mW	39,96 mW ⊗	13,66 mW	-
WAV	7,91 %	38,93 mW	37,63 mW	12,89 mW	10
MP3	- 0,01 %	43,02 mW	43,02 mW ⊗	15,4 mW	-
RAW	0,43 %	38,52 mW	38,45 mW	13,15 mW	192
BMP	4,95 %	35,11 mW	34,49 mW	11,27 mW	18
JPG	1,03 %	41,54 mW	41,34 mW	14,59 mW	74
TIFF	0,88 %	42,62 mW	42,45 mW	15,13 mW	85
PDF	- 1,16 %	42,82 mW	43,06 mW ⊗	15,39 mW	-

Tabela 5.9: Resultados obtidos com o esquema Transition em uma NoC Hermes com largura de *flit* igual a 16 bits.

Stream	Redução na transição de sinais	Pôtência média consumida na NoC sem codificação ($APB_F + APS_C + APL_R$)	Pôtência média consumida na NoC com codificação ($APB_F + APS_C + APL_R$)	Pôtência média consumida pelos módulos de codificação ($APE + APD$)	# of hops
HTML	- 2,22 %	37,15 mW	37,48 mW ⊗	11,77 mW	-
GZIP	- 0,4 %	43,16 mW	43,24 mW ⊗	13,51 mW	-
GCC	11,78 %	41,26 mW	39,05 mW	12,59 mW	6
Bytecode	- 2,42 %	37,9 mW	38,27 mW ⊗	11,99 mW	-
WAV	19,67 %	38,93 mW	35,7 mW	11,74 mW	4
MP3	0,05 %	42,98 mW	42,97 mW	13,44 mW	1183
RAW	- 11,41 %	38,52 mW	40,34 mW ⊗	12,4 mW	-
BMP	0,08 %	42,94 mW	42,92 mW	13,43 mW	826
JPG	- 0,72 %	42,78 mW	42,92 mW ⊗	13,4 mW	-
TIFF	1,8 %	42,62 mW	42,26 mW	13,28 mW	37
PDF	1,8 %	40,95 mW	40,62 mW	12,79 mW	39

Novamente, os melhores resultados foram obtidos com o esquema *T-Bus-Invert*, também para a NoC com *flits* de 16 bits. Conforme mostra a Tabela 5.10, na maioria dos casos a redução na transição de sinais passou de 31% e a quantidade de *hops* necessários para amortizar o custo da codificação foi de apenas 3 (PALMA et al., 2007-b).

Tabela 5.10: Resultados obtidos com o esquema *T-Bus-Invert* em uma NoC Hermes com largura de *flit* igual a 16 bits.

Stream	Redução na transição de sinais	Pôtência média consumida na NoC sem codificação ($APB_F + APS_C + APL_R$)	Pôtência média consumida na NoC com codificação ($APB_F + APS_C + APL_R$)	Pôtência média consumida pelos módulos de codificação ($APE + APD$)	# of hops
HTML	18,47 %	37,16 mW	34,46 mW	15,53 mW	6
GZIP	32,17 %	43,16 mW	36,52 mW	17,46 mW	3
GCC	29,48 %	41,26 mW	35,74 mW	16,84 mW	3
Bytecode	25,39 %	37,9 mW	34 mW	15,72 mW	4
WAV	35,7 %	38,93 mW	33,07 mW	15,96 mW	3
MP3	31,83 %	43,02 mW	36,05 mW	17,42 mW	3
RAW	25,18 %	38,52 mW	34,5 mW	15,94 mW	4
BMP	31,53 %	42,94 mW	36,51 mW	17,39 mW	3
JPG	31,13 %	41,54 mW	35,63 mW	16,91 mW	3
TIFF	32,68 %	43,09 mW	36,38 mW	17,43 mW	3
PDF	32,38 %	42,82 mW	36,26 mW	17,34 mW	3

No estudo-de-caso utilizando uma NoC com largura de *flit* igual a 32 bits, o esquema *Bus-Invert* foi implementado somente na versão com *clusters* de 8 bits (no caso, 4 *clusters*). Os resultados obtidos com a utilização deste esquema são apresentados na Tabela 5.11. Conforme mostra a segunda coluna desta tabela, a redução da atividade de transição de sinais foi bastante elevada, chegando a 30% no melhor caso, com o tráfego "pdf", e amortizando o custo da codificação já no primeiro *hop*. Na

maioria dos casos a quantidade de *hops* necessários para a amortização foi de apenas 2. Somente com dois tipos de tráfego (“html” e “bytecode”) a redução não foi suficiente para reduzir o consumo de potência na NoC.

Tabela 5.11: Resultados obtidos com o esquema *Bus-Invert* com 4 *clusters* em uma NoC Hermes com largura de *flit* igual a 32 bits.

Stream	Redução na transição de sinais	Pôtência média consumida na NoC sem codificação ($APB_F + APS_C + APL_R$)	Pôtência média consumida na NoC com codificação ($APB_F + APS_C + APL_R$)	Pôtência média consumida pelos módulos de codificação ($APE + APD$)	# of hops
HTML	7,14 %	98,02 mW	105,66 mW ⊗	15,97 mW	-
GZIP	28,54 %	120,83 mW	109,48 mW	19,19 mW	2
GCC	22,49 %	104,55 mW	100,76 mW	16,83 mW	4
Bytecode	17,47 %	99,34 mW	99,69 mW ⊗	16,1 mW	-
WAV	27,2 %	90,79 mW	85,59 mW	14,77 mW	3
MP3	27,58 %	120,78 mW	110,35 mW	19,19 mW	2
RAW	26,12 %	109,63 mW	102,27 mW	17,56 mW	2
BMP	27,02 %	120,3 mW	110,48 mW	19,12 mW	2
JPG	27,61 %	120,3 mW	109,92 mW	19,12 mW	2
TIFF	26,93 %	120,33 mW	110,6 mW	19,13 mW	2
PDF	30,57 %	121,35 mW	107,94 mW	19,25 mW	1

Na *NoC* com flits de 32 bits os esquemas *Gray* e *Transition* reduziram a atividade de transição de sinais com a maioria dos tráfegos simulados, como mostra a Tabela 5.12 e a

Tabela 5.13. Porém, a quantidade de *hops* necessários para amortizar o custo da codificação foi elevada. Primeiro pelo fato de que a redução da atividade de transição foi insignificante. Segundo por que o consumo de potência dos módulos de codificação e decodificação foi elevado para ambos os esquemas, principalmente no caso do esquema *Gray*, passando de 40 mW na maioria dos casos.

Tabela 5.12: Resultados obtidos com o esquema Gray em uma NoC Hermes com largura de *flit* igual a 32 bits.

Stream	Redução na transição de sinais	Pôtência média consumida na NoC sem codificação ($APB_F + APS_C + APL_R$)	Pôtência média consumida na NoC com codificação ($APB_F + APS_C + APL_R$)	Pôtência média consumida pelos módulos de codificação ($APE + APD$)	# of hops
HTML	- 33,37 %	67,67 mW	77,8 mW ⊗	41,05 mW	-
GZIP	1,72 %	79,08 mW	78,36 mW	42,71 mW	59
GCC	- 0,07 %	51,85 mW	51,86 mW ⊗	21,14 mW	-
Bytecode	- 11,04 %	69,5 mW	73,05 mW ⊗	37,93 mW	-
WAV	3,34 %	64,06 mW	63,17 mW	30,41 mW	34
MP3	0,51 %	79,06 mW	78,84 mW	43,05 mW	202
RAW	1,91 %	73,48 mW	72,79 mW	38,19 mW	55
BMP	0,94 %	63,78 mW	63,53 mW	30,64 mW	123
JPG	1,16 %	76,92 mW	76,46 mW	41,14 mW	89
TIFF	0,06 %	77,5 mw	77,47 mW	41,92 mW	1627
PDF	0,86 %	79,34 mW	78,98 mW	43,18 mW	120

Tabela 5.13: Resultados obtidos com o esquema Transition em uma NoC Hermes com largura de *flit* igual a 32 bits.

Stream	Redução na transição de sinais	Pôtência média consumida na NoC sem codificação ($APB_F + APS_C + APL_R$)	Pôtência média consumida na NoC com codificação ($APB_F + APS_C + APL_R$)	Pôtência média consumida pelos módulos de codificação ($APE + APD$)	# of hops
HTML	- 0,37 %	67,67 mW	67,79 mW ⊗	22,41 mW	-
GZIP	1,14 %	78,86 mW	78,39 mW	25,59 mW	54
GCC	3,59 %	51,85 mW	51,33 mW	17,67 mW	34
Bytecode	3,51 %	69,5 mW	68,37 mW	22,75 mW	20
WAV	- 5,69 %	72,73 mW	74,74 mW ⊗	24,18 mW	-
MP3	0,26 %	79,06 mW	78,95 mW	25,71 mW	239
RAW	- 11,19 %	73,48 mW	77,52 mW ⊗	24,71 mW	-
BMP	- 0,33 %	78,81 mW	78,95 mW ⊗	25,67 mW	-
JPG	0,16 %	78,81 mW	78,75 mW	25,64 mW	375
TIFF	- 0,12 %	77,5 mW	77,55 mW ⊗	25,28 mW	-
PDF	1,95 %	79,34 mW	78,52 mW	25,68 mW	31

Tabela 5.14: Resultados obtidos com o esquema *T-Bus-Invert* em uma NoC Hermes com largura de *flit* igual a 32 bits.

Stream	Redução na transição de sinais	Pôtência média consumida na NoC sem codificação ($APB_F + APS_C + APL_R$)	Pôtência média consumida na NoC com codificação ($APB_F + APS_C + APL_R$)	Pôtência média consumida pelos módulos de codificação ($APE + APD$)	# of hops
HTML	12,46 %	67,67 mW	63,89 mW	31,28	8
GZIP	27,39 %	79,7 mW	68,1 mW	35,1 mW	3
GCC	22,46 %	70,94 mW	63,39 mW	32,2 mW	4
Bytecode	20,01 %	68,33 mW	62,12 mW	31,35 mW	5
WAV	22,96 %	72,06 mW	64,08 mW	32,58 mW	4
MP3	27,09 %	79,06 mW	67,75 mW	34,88 mW	3
RAW	22,89 %	73,48 mW	65,2 mW	33,07 mW	4
BMP	26,51 %	78,81 mW	67,82 mW	34,82 mW	3
JPG	27,06 %	78,82 mW	67,59 mW	34,8 mW	3
TIFF	27,2 %	78,83 mW	67,54 mW	34,8 mW	3
PDF	28,17 %	79,34 mW	67,5 mW	34,95 mW	3

A Tabela 5.15 resume os resultados da comparação entre os esquemas *Adaptive Probability Encoding*, *Bus-Invert*, *Gray*, *Transition* e *T-Bus-Invert*, implementados para uma NoC com largura de *flit* igual a 8 bits. Os resultados são apresentados em termos de redução na atividade de transição de sinais e quantidade de *hops* necessários para amortizar o custo da codificação. Com base nesta comparação, se pode observar que o esquema *T-Bus-Invert* obteve os melhores resultados para todos os tráfegos simulados (PALMA et al., 2007-c).

A mesma comparação é feita para uma NoC com *flits* de 16 bits e apresentada na Tabela 5.16. Novamente o esquema *T-Bus-Invert* foi mais eficiente do que os outros esquemas implementados, com todos os tráfegos simulados, apresentado resultados ainda melhores do que na implementação com 8 bits (PALMA et al., 2007-c).

A Tabela 5.17 apresenta a comparação entre os esquemas implementados para a NoC com *flits* de 32 bits. Os resultados do esquema *T-Bus-Invert* foram bastante parecidos com os resultados da implementação com 16 bits. Entretanto, o esquema mais eficiente para esta configuração da NoC foi o *Bus-Invert* com 4 *clusters* de 8 bits, exceto para os tráfegos "html" e "bytecode". Para estes dois tipos de tráfego o melhor esquema foi, novamente, o *T-Bus-Invert*.

Tabela 5.15: Comparação entre os resultados obtidos com diferentes esquemas de codificação em uma NoC Hermes com largura de *flit* igual a 8 bits.

Stream	Adaptive Encoding		Bus-Invert		Gray		Transition		T-Bus-Invert	
	Transition Reduction	# of hops	Transition Reduction	# of hops	Transition Reduction	# of hops	Transition Reduction	# of hops	Transition Reduction	# of hops
HTML	-1,4 %	-	6,2 %	-	- 11,36 %	-	- 2,97 %	-	9,8 %	12
GZIP	1,03%	240	18,7%	-	- 0,49 %	-	1,17 %	61	26,89 %	3
GCC	0,91 %	292	17,9%	-	- 5,57 %	-	2,38 %	31	26,35 %	4
Bytecode	9,3 %	32	12 %	-	0,21 %	331	9,22 %	9	20,88 %	5
WAV	21,95 %	11	18,8 %	-	- 0,01 %	-	12,22 %	6	28,19 %	3
MP3	- 2,41 %	-	18,48 %	-	- 0,06 %	-	- 0,21 %	-	27,09 %	3
RAW	- 10,98 %	-	14,6 %	-	4,11 %	19	- 12,44 %	-	14,5 %	8
BMP	- 0,5 %	-	18,2%	-	11,82 %	7	- 0,02 %	-	26,38 %	4
JPG	0,8 %	327	19,5 %	-	3,42 %	19	- 0,6 %	-	26,26 %	4
TIFF	-1,16 %	-	18,3 %	-	- 1,36 %	-	- 1,16 %	-	27,68 %	3
PDF	6,61 %	36	23,15 %	13	3,5 %	18	7,52 %	9	30,57 %	3

Tabela 5.16: Comparação entre os resultados obtidos com diferentes esquemas de codificação em uma NoC Hermes com largura de *flit* igual a 16 bits.

Stream	Bus-Invert 2 clusters		Bus-Invert 1 cluster		Gray		Transition		T-Bus-Invert	
	Transition Reduction	# of hops	Transition Reduction	# of hops	Transition Reduction	# of hops	Transition Reduction	# of hops	Transition Reduction	# of hops
HTML	11,73 %	-	10,57 %	-	- 32,07 %	-	- 2,22 %	-	18,47 %	6
GZIP	22,83 %	17	19,59 %	6	- 0,45 %	-	- 0,4 %	-	32,17 %	3
GCC	20 %	-	16,86 %	13	- 4,09 %	-	11,78 %	6	29,48 %	3
Bytecode	17,08 %	-	12,89 %	-	- 13,42 %	-	- 2,42 %	-	25,39 %	4
WAV	29,04 %	6	22,69 %	5	7,91 %	10	19,67 %	4	35,7 %	3
MP3	23,14 %	15	19,9 %	6	- 0,01 %	-	0,05 %	1183	31,83 %	3
RAW	21,41 %	-	17,2 %	16	0,43 %	192	- 11,41 %	-	25,18 %	4
BMP	22,81 %	18	19,19 %	6	4,95 %	18	0,08 %	826	31,53 %	3
JPG	23,27 %	15	19,99 %	6	1,03 %	74	- 0,72 %	-	31,13 %	3
TIFF	23,33 %	15	20,75 %	5	0,88 %	85	1,8 %	37	32,68 %	3
PDF	23,99 %	11	21,52 %	4	- 1,16 %	-	1,8 %	39	32,38 %	3

Tabela 5.17: Comparação entre os resultados obtidos com diferentes esquemas de codificação em uma NoC Hermes com largura de *flit* igual a 32 bits.

Stream	Bus-Invert 4 clusters		Gray		Transition		T-Bus-Invert	
	Transition Reduction	# of hops	Transition Reduction	# of hops	Transition Reduction	# of hops	Transition Reduction	# of hops
HTML	7,14 %	-	- 33,37 %	-	- 0,37 %	-	12,46 %	8
GZIP	28,54 %	2	1,72 %	59	1,14 %	54	27,39 %	3
GCC	22,49 %	4	- 0,07 %	-	3,59 %	34	22,46 %	4
Bytecode	17,47 %	-	- 11,04 %	-	3,51 %	20	20,01 %	5
WAV	27,2 %	3	3,34 %	34	- 5,69 %	-	22,96 %	4
MP3	27,58 %	2	0,51 %	202	0,26 %	239	27,09 %	3
RAW	26,12 %	2	1,91 %	55	- 11,19 %	-	22,89 %	4
BMP	27,02 %	2	0,94 %	123	- 0,33 %	-	26,51 %	3
JPG	27,61 %	2	1,16 %	89	0,16 %	375	27,06 %	3
TIFF	26,93 %	2	0,06 %	1627	- 0,12 %	-	27,2 %	3
PDF	30,57 %	1	0,86 %	120	1,95 %	31	28,17 %	3

5.2 Conclusão

Este capítulo apresentou a análise do consumo de potência da NoC com e sem codificação, apresentando resultados experimentais obtidos através de simulações com diferentes tipos de tráfego. A análise foi feita com base nos macromodelos de potência apresentados no capítulo 4. Dentre os esquemas de codificação implementados, o que apresentou os melhores resultados foi o *T-Bus-Invert*, proposto neste trabalho. O *T-Bus-Invert* reduziu significativamente a atividade de transição de sinais e, conseqüentemente, o consumo de potência na NoC para todos os estudos-de-caso apresentados.

6 CONCLUSÕES

Este trabalho investigou a redução do consumo de potência em Networks-on-Chip através da redução da atividade de transição de sinais utilizando esquemas de codificação de dados. Macromodelos de consumo de potência para os módulos da NoC, bem como para os módulos de codificação e decodificação foram desenvolvidos e embarcados em um modelo de mais alto nível. Este modelo de mais alto nível foi simulado com tipos de tráfego real, para que se pudesse avaliar a eficiência dos esquemas de codificação em diferentes configurações da NoC Hermes.

Diferentes esquemas de codificação encontrados na literatura foram implementados, simulados e avaliados, juntamente com um novo esquema de codificação, o *T-Bus-Invert*, proposto no escopo deste trabalho.

Os resultados obtidos mostram que a eficiência do esquema de codificação é dependente do padrão de dados transmitido. Os módulos do esquema *Adaptive Probability Encoding* apresentaram um consumo elevado de potência quando comparados ao consumo da NoC e dos demais esquemas de codificação. Além disso, apresentou resultados insatisfatórios quanto à redução da atividade de transição de sinais. Os módulos do esquema *Transition* são econômicos quanto ao consumo de potência. Porém, este esquema foi eficiente em pouquíssimos casos, e utilizando *flits* com largura 8 e 16. Neste trabalho, o esquema é dito eficiente quando reduz significativamente a atividade de transição de sinais, reduzindo assim o consumo médio de potência da transmissão dos dados, levando-se em consideração o consumo extra de potência nos módulos de codificação e de decodificação.

O esquema *Gray* não apresentou bons resultados quanto à redução da atividade de transição de sinais. Além disso, o módulo de decodificação apresenta um consumo elevado de potência com larguras de *flit* maiores do que 16 bits. Os módulos do esquema *Bus-Invert* consomem pouca potência quando comparados à NoC e aos módulos dos outros esquemas de codificação, mas não são eficientes com *flits* de 8 bits, nem na versão de 16 bits separados em 2 *clusters* de 8 bits. Na versão com 1 *cluster* de 16 bits o esquema já se torna eficiente, mas na versão de 32 bits separados em 4 *clusters* de 8 bits é que foram encontrados os melhores resultados. Neste caso o esquema *Bus-Invert* é o mais eficiente dentre os comparados neste trabalho. O esquema *T-Bus-Invert* reduziu a mesma percentagem da atividade de transição de sinais nas 3 versões (com 8, 16 e 32 bits), apresentando os melhores resultados para a NoC Hermes com larguras de *flit* igual a 8 e 16 bits. Porém, foi superado pelo *Bus-Invert* na NoC com *flits* de 32 bits devido à redução significativa na atividade de transição obtida com este esquema.

O esquema *T-Bus-Invert* apresenta como vantagens o fato de utilizar a técnica eficiente do método *Bus-Invert* sem necessitar que a NoC seja alterada. Sua desvantagem é o fato de que ele reduz o throughput máximo da NoC. Porém, como comentado na seção 3.7, a latência inserida por este esquema não chega a prejudicar o desempenho da aplicação-alvo.

É importante salientar que os resultados apresentados neste trabalho foram obtidos após a simulação elétrica dos módulos utilizando-se a tecnologia 0.35µm. Em ambos os esquemas utilizados, a redução do consumo de potência nos canais entre roteadores foi muito menor do que na lógica do roteador. Entretanto, em novas technologias, o consumo de potência nos canais será mais relevante, devido ao crescimento relativo das capacitâncias nas linhas de comunicação em comparação com a lógica (SYLVESTER; CHENMING, 2001; SYLVESTER; KEUTZER, 2001). Neste cenário tais esquemas de codificação podem ser mais vantajosos, já que eles foram desenvolvidos para canais de comunicação.

Como trabalhos futuros podemos citar a avaliação dos esquemas de codificação em NoCs implementadas utilizando tecnologias do estado-da-arte. Além disso, outros esquemas de codificação podem ser analisados quando implementados em sistemas baseados em NoCs. Como exemplos de outros esquemas de codificação não abordados neste trabalho podemos citar o *Partial Bus-Invert* (YOUNGSOO; SOO-IK; KIYOUNG, 1998), o *xor-pbm* e o *dbm-pbm* (RAMPRASAD; SHANBHAG, 1999). Existem também esquemas de codificação, como o *Odd/Even Bus-Invert*, que são direcionados à redução de potência levando em consideração as capacitâncias de acoplamento (ZHANG; LACH; SKADRON; STAN, 2002), um problema crescente em novas tecnologias (SYLVESTER; CHENMING, 2001).

Também como trabalhos futuros se pode analisar o uso de múltiplos esquemas para melhorar a eficiência da codificação em diferentes padrões de tráfego, bem como o uso de diferentes configurações da NoC, para ajudar na decisão de quando um pacote deve ser ou não codificado. Por exemplo, não é vantajoso codificar pacotes enviados a núcleos vizinhos na NoC. Uma possível abordagem é utilizar um bit de sinalização no cabeçalho do pacote, indicando se ele é ou não codificado.

REFERÊNCIAS

ARM. **AMBA 2.0 Specification**. Disponível em: <http://www.arm.com/armtech/ AMBA_Spec >. Acesso em: set. 2003.

ARVIND, K.; NIKHIL, R. S. **Executing a Program on the MIT Tagged Token Dataflow Architecture**. Eindhoven: Springer-Verlag, 1987. p.1-29. (Lecture Notes in Computer Science, v.2).

ASHRAF, F.; ABD-EL-BARR, M.; AL-TAWIL, K. Introduction to Routing in Multicomputer Networks. **Computer Architecture News**, New York, v.26, n.5, p.14-21, Dec. 1998.

BEECROFT, J.; HOMEWOOD, M.; MCLAREN, M. Meiko CS-2 Interconnect Elan-Elite Design. **Parallel Computing**, Netherlands, v.20, n.10-11, p.1627-1638, Nov. 1994.

BENINI, L.; DE MICHELI, G. Networks on chips: a new SOC paradigm. **Computer**, Los Alamitos, CA, v.35, n.1, p. 70-78, Jan. 2002.

BENINI, L.; DE MICHELI, G. Powering Networks on Chips. In: INTERNATIONAL SYMPOSIUM ON SYSTEM SYNTHESIS, 14., 2001, Montreal, **Proceedings...** Los Alamitos: IEEE Computer Society , 2002. p.33-38.

BENINI, L. et al. Architecture and Synthesis Algorithms for Power-Efficient Bus Interfaces. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, New York, v.19, n.9, p. 969-980, Sept. 2000.

BENINI, L. et al. Address Bus Encoding Techniques for System-Level Power Optimization. In: DATE, 1998. **Proceedings…** [S.l.: s.n.], 1998. p. 861-866.

BENINI, L. et al. Reducing Power Consumption of Core-Based Systems By Address Bus Encoding. **IEEE Trans. on VLSI Systems**, New York, v.6, n.4, p. 554-562, Dec. 1998.

BERGAMASCHI, R. A.; LEE, W. R. Designing Systems-on-Chip Using Cores. In: DESIGN AUTOMATION CONFERENCE, DAC, 37., 2000. **Proceedings...** New York: ACM Press, 2000. p.420-425.

BODEN, N. J. et al. Myrinet: a Gigabit-per-second Local Area Network. **IEEE Micro**, Los Alamitos, v.15, n.1, p.29-36, Feb. 1995.

BROOKS C. et al. **Heterogeneous Concurrent Modeling and Design in Java**. Berkeley, CA: University of California, 2005. v.1.

BURD, T.; BRODERSEN, R. **Energy Efficient Microprocessor Design**. Boston: Kluwer Academic Publishers, 2002. 376 p.

CALAZANS, N. **Projeto Lógico Automatizado de Sistemas Digitais Seqüenciais**. Rio de Janeiro: DCC/IM, 1998. 318 p. Trabalho apresentado na 11a. Escola de Computação, 1998.

CHANDRAKASAN, A.; BRODERSEN R. **Low Power Digital CMOS Design**. Boston: Kluwer Academic Publishers, 1995.

COSTA, E. A. C. da. **Operadores Aritméticos de Baixo Consumo para Arquiteturas de Circuitos DSP**. 2002. 193 p. Tese (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

CULLER, D.; GUPTA, A.; SINGH, J. P. **Parallel Computer Architecture**: a Hardware Software Approach. Los Altos, California : Morgan Kaufmann, 1998. 1100p.

DALLY, W.; TOWLES, B. Route packets, not wires: on-chip interconnection networks. In: DESIGN AUTOMATION CONFERENCE, DAC, 38., 2001, Las Vegas. **Proceedings...** New York: ACM, 2001. p.684-689.

DALLY, W. J.; SEITZ, C. L. Deadlock-Free Message Routing in Multiprocessors Interconnection Networks. **IEEE Transactions on Computers**, New York, v.C-36, n.5, p.547-553, May 1987.

DALLY, W. J.; SEITZ, C. L. The Torus Routing Chip. **Journal of Distributed Computing**, [S.l.], v.1, n.3, p.187-196, Oct. 1986.

DAY, J. D.; ZIMMERMMAN, H. The OSI reference model. **Proceedings of IEEE**, New York, v.71, n.12, p. 1334-1340, Dec. 1983.

DESIGN-REUSE. Disponível em: < http://www.design-reuse.com >. Acesso em: set. 2003.

DUATO, J.; YALAMANCHILI, S.; NI, L. **Interconnection Networks**: An Engineering Approach . Los Alamitos, California : IEEE Computer Society Press, 1997. 515p.

FENG, T.-Y. A Survey of Interconnection Networks. **Computer**, Los Alamitos, CA, v.14, n.12, p. 12-27, Dec. 1981.

GALLES, M. Spider: a High Speed Network Interconnect. **IEEE Micro**, Los Alamitos, CA, v.17, n.1, p.34-39, Jan.-Feb. 1997.

GLASS, C.; NI, L. The Turn Model for Adaptive Routing. **Journal of the Association for Computing Machinery**, New York, v.41, n.5, p. 874-902, Sept. 1994.

GUERRIER, P. **Un Réseau d'Interconnexion pour Systèmes Intégrés**. 2000. Tese (Doutorado) – Université Pierre et Marie Curie, Paris. Disponível em: < http://www-asim.lip6.fr/~adrijean >. Acesso em: set. 2003.

GUERRIER, P.; GREINER, A. A generic architecture for on-chip packet-switched interconnections. In: DESIGN AUTOMATION AND TEST IN EUROPEAN CONFERENCE AND EXHIBITION, DATE, 2000. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 2000. p.250-256.

GUERRIER, P.; GREINER, A. A Scalable Architecure for System-on-Chip Interconnections. In: SOPHIA-ANTIPOLIS MICRO-ELECTRONICS CONFERENCE, 1999, France. **Proceedings…** Sophia Antipolis : [s.n.], 1999. p.90-93.

GUPTA, R. K., ZORIAN, Y. Introducing Core-Based System Design. **IEEE Design & Test of Computers**, Los Alamitos, CA, v.14, n.4, p. 15-25, Oct.-Dec. 1997.

GURD, J.; KIRKHAM, C.; WATSON, I. The Manchester Prototype Dataflow Computer. **Communications of the ACM**, New York, v.28, n.1, p.34-53, Jan. 1985.

GURD, J.; SNELLING, D. The Manchester Dataflow Computing System. In: DONGARRA, J.J. (Ed.). **Experimental Parallel Computing Systems**. [S.l.]: North-Holland, 1987. p.177-219.

HU, J.; MARCULESCU, R. Energy-aware mapping for tile-based NoC architectures under performance constraints. In: ASIA AND SOUTH PACIFIC-DESIGN AUTOMATION CONFERENCE, ASP-DAC, 2003. **Proceedings...** Piscataway, NJ: IEEE, 2003. p.233-239.

HU, J.; DENG, Y.; MARCULESCU, R. System-Level Point-to-Point Communication Synthesis Using Floorplanning Information. In: INT. CONFERENDE ON VLSI DESIGN, 15., 2002, Bangalore. **Proceedings...** Los Alamitos: IEEE Computer Society, 2002. p.297-301.

HWANG, K. **Advanced Computer Architecture**: Parallelism, Scalability, Programmability. New York : McGraw-Hill, 1993. 770p.

IBM. **The RS/6000 SP High Performance Communication Network**. Disponível em: < http://www.rs6000.ibm.com/resource/technology/sp_sw1/ spswp1.book_1.html >. Acesso em: ago. 1999.

IBM. **Interconnection Technologies for High-Performance Computing** (RS/6000 SP). IBM POWERparallel Technology Briefing. Disponível em: < http://www. rs6000.ibm.com/resource/technology/sp_sw2/spswp2_1.html >. Acesso em: ago. 1999.

IBM. **The CoreConnectTM Bus Architecture**. Disponível em: <http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF7785256991004DB5D9/$file/crcon_pb.pdf>. Acesso em: jul. 2007.

IYER, A.; MARCULESCU, D. Power and performance evaluation of globally asynchronous locally synchronous processors. ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, ISCA, 29., 2002, Archorage, Alasca. **Proceedings...** Los Alamitos, CA: IEEE, 2002. p.158-168.

ITRS - International Technology Roadmap for Semiconductors, 2005. Disponível em: < http://www.itrs.net/Common/2005ITRS/Design2005.pdf >. Acesso em: mar. 2006.

JANTSCH, A.; TENHUNEM, H. Will Networks-on-Chip Close the Productivity Gap? In: JANTSCH, A.; TENHUNEM, H. (Ed.). **Networks On Chip**. Boston: Kluwer Academic Publishers, 2003. p.3-18.

KERMANI, P.; KLEINROCK, L. Virtual Cut-Through: A New Computer Communication Switching Technique. **Computer Network**, [S.l.], v.3, n.4, p.267-286, Sept. 1979.

KESSLER, R. E.; SCHWARZMEIER, J. L. Cray T3D: a New Dimension for Cray Research. In: COMPCON, 1993, San Francisco. **Proceedings…** Los Alamitos : IEEE Computer Society, 1993. p.176-82.

KONSTANTINIDOU, S.; SNYDER, L. The Chaos Router: Architecture and Performance. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 18., 1991, Toronto. **Proceedings…** New York : ACM Press, 1991. p.79-88.

KORNAROS, G. et al. ATLAS I: Implementing a Single-Chip ATM Switch with Backpressure. **IEEE Micro**, Los Alamitos, CA, v.19, n.1, p.30-41, Jan.-Feb. 1999.

KUMAR, S. et al. A Network on Chip Architecture and Design Methodology. In: INT. SYMPOSIUM ON VERY LARGE INTEGRATION SCALE, 2002, Pittsburg. **Proceedings...** Los Alamitos: IEEE Computer Society, 2002. p.105-112.

LANGEN, D.; BRINKMANN, A.; RUCKERT, U. High Level Estimation of the Area and Power Consumption of on-Chip Interconnects. In: ASIC/SOC CONFERENCE, 13., 2000, Arlington. **Proceedings...** Piscataway : IEEE, 2000. p.297-301.

LAUDON, J.; LENOSKI; D. The SGI Origin: A ccNUMA Highly Scalable Server. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 24.,1997, Denver. **Proceedings…** New York : ACM Press, 1997. p.241-251.

LIANG, J.; SWAMINATHAN, S.; TESSIER, R. aSOC: A Scalable, Single-Chip Communication Architecture. In: INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, 2000, Philadelphia. **Proceedings...** Los Alamitos: IEEE Computer Society, 2000. Disponível em: < http://www-unix.ecs.umass.edu/~jliang/web/ publications.html >. Acesso em: set. 2003.

MADISETTI, V. K.; SHEN, L. Interface Design for Core-Based Systems. **IEEE Design & Test of Computers**, Los Alamitos, CA, v.14, n.4, p. 42-51, Oct.-Dec. 1997.

MARCON, C.; CALAZANS, N.; MORAES, F.; SUSIN, A. REIS, L.; HESSEL, F. Exploring NoC Mapping Strategies: An Energy and Timing Aware Technique. In: DESIGN, AUTOMATION AND TEST IN EUROPE CONFERENCE AND EXHIBITION, DATE, 2005, Munich. **Proceedings...** [S.l.]: IEEE, 2005. v.1.

MARCON, C.; BORIN, A.; SUSIN, A; CARRO, L.; WAGNER, F. Time and Energy Efficient Mapping of Embedded Applications onto NoCs. ASIA AND SOUTH

PACIFIC-DESIGN AUTOMATION CONFERENCE, ASP-DAC, 2005. **Proceedings...** Piscataway, NJ: IEEE, 2005 (a). v.1. p.33-38.

MARCON, C.; PALMA, J.; CALAZANS, N.; SUSIN, A.; REIS, R.; MORAES, F. Modeling the Traffic Effect for the Application Cores Mapping Problem onto NoCs. In: IFIP INTERNATIONAL CONFERENCE ON VERY LARGE SCALE INTEGRATION, VLSI-SOC, 2005, Perth. **Proceedings...** New York: Springer, 2007. p.179-194.

MATTSON, T. G.; HENRY, G. **An Overview of the Intel TFLOPS Supercomputer.** 1998. Disponível em: < http://developer.intel.com/technology/itj/q11998/articles/art_1.htm >. Acesso em: set. 2003.

MEHTA, H.; OWENS, R.; IRWIN, M. Some Issues in Gray Code Addressing. In: GLS-VLSI, 1996. **Proceedings...** [S.l.: s.n.], 1996. p.178-180.

MELLO, A. et al. MAIA - A Framework for Networks on Chip Generation and Verification. In: ASIA AND SOUTH PACIFIC-DESIGN AUTOMATION CONFERENCE, ASP-DAC, 2005. **Proceedings...** Piscataway, NJ: IEEE, 2005 (a). v.1.

MELLO, A.; MÖLLER, L. **Arquitetura Multiprocessada em SoCs**: Estudo de Diferentes Topologias de Interconexão. 2003. Trabalho de Conclusão de Curso. FACIN, PUCRS, Porto Alegre.

MOHAPATRA, P. Wormhole Routing Techniques for Directly Connected Multicomputer Systems. **ACM Computing Surveys**, New York, v.30, n.3, p. 374, Sept. 1998.

MORAES, F. et al. HERMES: an infrastructure for low area overhead packet-switching networks on chip. **The VLSI Journal Integration (VJI)**, [S.l.],v.38, n.1, p. 69-93, Oct. 2004.

MURALI, S.; DE MICHELI, G. Bandwidth-constrained mapping of cores onto NoC architectures. In: DESIGN, AUTOMATION AND TEST IN EUROPE CONFERENCE AND EXHIBITION, DATE, 2004. **Proceedings...** Los Alamitos: IEEE, 2004. p. 896-901.

MUSOLL, E.; LANG, T.; CORTADELLA, J. Working-Zone Encoding for Reducing the Energy in Microprocessor Address Busses. **IEEE Trans.on VLSI Systems**, New York, v.6, n.4, p. 568-572, Dec. 1998.

NI, L.; McKINLEY, P. A Survey of Wormhole Routing Techniques in Direct Networks. **IEEE Computer Magazine**, [S.l.], v.26, n.2, p.62-76, Feb. 1993.

NUGENT, S. The iPSC/2 Direct Connect Communications Technology. In: CONFERENCE ON HYPERCUBE CONCURRENT COMPUTERS AND APPLICATIONS, 3., 1988, Pasadena. **Proceedings…** New York : ACM Press, 1988. p.51-59.

OPENCORES.ORG. **Wishbone SoC Interconnection**. Disponível em: < http://www.opencores.com/press/pr_8jan2001.shtml >. Acesso em: set. 2003.

OPENCORES.ORG. **Plasma - most MIPS I(TM) opcodes**. Disponível em: < http://www.opencores.org/ projects.cgi/ web/mips/overview >. Acesso em: ago. 2007.

PALMA, J. **Técnicas para Implementação de Sistemas Digitais Reutilizáveis**. 2000. Trabalho Individual (Mestrado em Ciência da Computação) – Programa de Pós-Graduação em Ciência da Computação, PUCRS, Porto Alegre.

PALMA, J. **Métodos de Distribuição e Conexão de IC Cores para Dispositivos Programáveis FPGA**. 2002. 108f. Dissertação de Mestrado (Mestrado em Ciência da Computação) – Programa de Pós-Graduação em Ciência da Computação, PUCRS, Porto Alegre.

PALMA, J. **Um Estudo Sobre Redes de Conexão Intra-Chip**. 2003. Trabalho Individual (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

PALMA, J.; MARCON, C.; MORAES, F.; CALAZANS, N.; REIS, R.; SUSIN, A. Mapping Embedded Systems onto NoCs - The Traffic Effect on Dynamic Energy Estimation. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 18., 2005. **Proceedings…** New York: ACM Press, 2005. p. 196-201.

PALMA, J.; INDRUSIAK, L.; MORAES, F.; ORTIZ, A.; GLESNER, M.; REIS, R. Evaluating the Impact of Data Encoding Techniques on the Power Consumption in Networks-on-Chip In: IEEE COMPUTER SOCIETY ANNUAL SYMPOSIUM ON EMERGING VLSI TECHNOLOGIES AND ARCHITECTURES, 2006, Karlsruhe, Germany. **Proceedings...** Los Alamitos, CA: IEEE Computer Society, 2006.

PALMA, J.; INDRUSIAK, L.; MORAES, F.; ORTIZ, A.; GLESNER, M.; REIS, R. Adaptive Coding in Networks-on-Chip: Transition Activity Reduction versus Power Overhead of the Codec Circuitry. In: PATMOS, 16., 2006, Montpellier, França. **Proceedings...** Berlin: Springer, 2006. p. 603-613.

PALMA, J.; INDRUSIAK, L.; MORAES, F.; ORTIZ, A.; GLESNER, M.; REIS, R. Avaliando o Impacto de Técnicas de Codificação de Dados Sobre o Consumo de Potência em Networks-On-Chip. In: WORKSHOP IBERCHIP, 13., 2007, Lima. **Proceedings...** Lima: Hozlo S.R.L., 2007. p. 413-418.

PALMA, J.; INDRUSIAK, L.; MORAES, F.; ORTIZ, A.; GLESNER, M.; REIS, R. Inserting Data Encoding Techniques into NoC-Based Systems. In: IEEE COMPUTER SOCIETY ANNUAL SYMPOSIUM ON VLSI, 2007, Porto Alegre. **Proceedings...** Los Alamitos, CA: IEEE, 2007. p.299-304.

PALMA, J.; INDRUSIAK, L.; MORAES, F.; GLESNER, M.; REIS, R. **Reducing the Power Consumption in Networks-on-Chip through Data Coding Schemes**. Aceito para publicação no 14th IEEE International Conference on Electronics, Circuits and Systems, ICECS 2007, Marrakech, Marrocos.

PEDRAM, M. Power minimization in IC design. **ACM Trans. Design Automat. Electron. Syst**., New York, v.1, n.1, p.3-58, Jan. 1996.

RABAEY, J. M. **Digital Integrated Circuits**: A Design Perspective. Upper Saddle River : Prentice Hall, 1996.702p.

RAMPRASAD, S.; SHANBHAG, N. A Coding Framework for Low-Power Address and Data Busses. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, Princeton, NJ, v.7, n.2, p.212-222, June 1999.

RAMOS, P.; OLIVEIRA, A. Low Overhead Encodings for Reduced Activity in Data and Address Buses. In: INTERNATIONAL SYMPOSIUM ON SIGNALS, CIRCUITS AND SYSTEMS. **Proceedings…** [S.l.: s.n.], 1999. p. 21-24.

RIJPKEMA, E. A Router Architecture for Network on Silicon. In: WORKSHOP PROGRESS, 2001. **Proceedings...** Netherlands: [s.n.], 2001.

RINCON, A. et al. Core Design and System-on-a-chip Integration. **IEEE Design & Test of Computers**, Los Alamitos, CA, p. 26-35, Oct.-Dec. 1997.

SCHALLER, R. R. Moore´s Law: Past, Present and Future. **IEEE Spectrum**, New York, v.34, n.6, p. 52-59, June 1997.

SEITZ, C. L.; SU, W. A Family of Routing and Communication Chips Based on the Mosaic. In: WASHINGTON SYMPOSIUM OF INTEGRATED SYSTEMS, 1993. **Proceedings…** [S.l.: s.n.], 1993. p.320-337.

SGI. **SGI - Cray T3E:** Performance of the Cray T3E Multiprocessor. Disponível em: < http://www.sgi.com/t3e/performance.html >. Acesso em: ago. 1999.

SGROI, M. et al. Addressing the System-on-a-Chip Interconnect Woes Through Communication-Based Design. In: DESIGN AUTOMATION CONFERENCE, DAC, 38., 2001. **Proceedings...** New York: ACM, 2001. p.667-672.

SIA - Semiconductor Industry Association. **SIA Annual Report 2005**. Disponível em: < http://www.sia-online.org/downloads/SIA_AR_2005.pdf >. Acesso em: mar. 2006.

SILICORE. **The WISHBONE Service Center**. Disponível em: < http://www.silicore.net/wishbone.htm >. Acesso em: set. 2003.

SINGH, D. et al. Power conscious cad tool and methodologies: A perspective. **Proceedings of the IEEE**, Piscataway, NJ, v.83, n.4, p. 570-594, Apr. 1995.

SOININEN, J.; HEUSALA, H. A Design Methodology for NOC-Based Systems. Networks On Chip. In: JANTSCH, A.; TENHUNEM, H. (Ed.). **Networks On Chip**. Boston: Kluwer Academic Publishers, 2003.

STAN, M. R.; BURLESON, W. P. Bus-Invert Coding for Low-Power I/O. **IEEE Transactions on VLSI Systems**, Princeton, NJ, v.3, n.1, p.49-58, Mar. 1995.

STAN, M. R.; BURLESON, W. P. Limited Weight Codes for Low-Power I/O. In: INTERNATIONAL WORKSHOP ON LOW POWER DESIGN, 1997. **Proceedings...** [S.l.: s.n.], 1997. p.70-73.

STAN, M. R.; BURLESON, W. P. Low-Power Encodings for Global Communication in CMOS VLSI. **IEEE Transactions on VLSI Systems**, Princeton, NJ, v.5, n.4, p.444-455, Dec. 1997.

STUNKEL, C. B. et al. The SP-1 High Performance Switch. In: SCALABLE HIGH PERFORMANCE COMPUTING CONFERENCE, 1994. **Proceedings…**[S.l.: s.n.], 1994. p. 150-157.

SYLVESTER, D.; CHENMING, W. Analytical modeling and characterization of deep-submicrometer interconnect. **Proceedings of the IEEE**, Piscataway, NJ, v.89, n.5, p.634 – 664, May 2001.

SYLVESTER, D.; KEUTZER, K. Impact of small process geometries on microarchitectures in systems on a chip. **Proceedings of the IEEE**, Piscataway, NJ, v.89, n.4, p.467-489, April 2001.

SU, C.; DESPAIN, A. A Cache Design Tradeoffs for Power and Performance Optimization AQ Case Study. In: IEEE SYMPOSIUM ON LOW POWER DESIGN, 1995. **Proceedings…**[S.l.: s.n.], 1995. p.63-68.

TAMIR, Y.; FRAZIER, G. L. Dynamically-Allocated Multi-Queue Buffers for VLSI Communication Switches. **IEEE Transactions on Computers**, [S.l.], v.41, n.6, p.725-737, June 1992.

WESTE, N.H.; ESHRAGHIAN, K. **Principles of CMOS VLSI Design**: A Systems Perspective. 2nd ed. Reading: Addison-Wesley, 1993.

WINGARD, D. MicroNetwork-based integration for SOCs. In: DESIGN AUTOMATION CONFERENCE, DAC, 38., 2001. **Proceedings…**New York: ACM, 2001. p.673-677.

YE, T.; BENINI, L.; DE MICHELI, G. Packetization and routing analysis of on-chip multiprocessor networks. **Journal of Systems Architecture (JSA)**, [S.l.], v.50, n.2-3, p.81-104, Feb. 2004.

YE, T.; BENINI, L.; DE MICHELI, G. Analysis of power consumption on switch fabrics in network routers. In: DESIGN AUTOMATION CONFERENCE, DAC, 2002. **Proceedings…**New York: ACM, 2002. p.524-529.

YOUNGSOO, S.; SOO-IK, C.; KIYOUNG, C. Partial bus-invert coding for power optimization of system level bus. In: INTERNATIONAL SYMPOSIUM ON LOW POWER ELECTRONICS AND DESIGN, 1998. **Proceedings…**New York: ACM, 1998. p.127-129.

ZEFERINO, C. A. **Redes de Interconexão para Multiprocessadores**. 1999. 123f. Trabalho Individual (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre. Disponível em: < http://www.inf.ufrgs.br/~zeferino/ Qualify_Zeferino_Redes_de_interconexao_para_ multiprocessadores.PDF >. Acesso em: abr. 2003.

ZEFERINO, C. A. **Redes-em-Chip**: Arquiteturas e Modelos para Avaliação de Área e Desempenho. 2003. 242f. Tese (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

ZHANG, Y. et al. Odd/Even Bus-Invert with Two-Phase Transfer for Buses with Coupling. In INTL. SYMP. ON LOW POWER ELECTRONICS AND DESIGN, ISLPED, 2002. **Proceedings…**[S.l:s.n.], 2002. p.80-83.

ANEXO CÓDIGOS VHDL

As tabelas A.1 a A.19 apresentam os códigos VHDL de alguns dos módulos de codificação e de decodificação implementados neste trabalho. Os códigos não apresentados são semelhantes aos que se encontram no anexo, variando apenas a largura de *flit*.

Tabela A.1: Código VHDL do codificador *Gray* com largura de *flit* igual a 8 bits.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity encoder_gray is
port(
   clock_tx_in : in std_logic;
   reset : in std_logic;
   data_in : in std_logic_vector(7 downto 0);
   rx : in std_logic;
   clock_tx_out : out std_logic;
   data_out : out std_logic_vector(7 downto 0);
   tx : out std_logic;
   credit_i_in : in std_logic;
   credit_i_out : out std_logic
);
end encoder_gray;

architecture encoder_gray of encoder_gray is

--tamanho do pacote; segundo flit do cabecalho
signal tamanho_pacote : std_logic_vector(7 downto 0);
signal cont_flit_payload, data_cod : std_logic_vector(7 downto 0);
type fsm is (S0,S1,s2);
signal EA : fsm;
signal func_xor : std_logic_vector(6 downto 0);

begin
   process(reset, clock_tx_in)
   begin
      if reset='1' then
         EA <= S0;
         cont_flit_payload <= (others=>'0');
      elsif clock_tx_in'event and clock_tx_in='0' then
         if credit_i_in='1' then
            case EA is
               -- estado inicial, antes de uma transmissao
               when S0 =>
                  if rx ='1' then
                     EA <= S1;
                  else
                     EA <= S0;
                  end if;
               -- inicio de transmissao, transmitindo cabeçalho
```

```
                when S1 =>
                   if rx ='1' then
                      tamanho_pacote <= data_in;
                      cont_flit_payload <= cont_flit_payload + 1;
                      EA <= S2;
                   else
                      EA <= S1;
                   end if;
                when S2 =>
                   cont_flit_payload <= cont_flit_payload + 1;
                   if cont_flit_payload = tamanho_pacote then
                      cont_flit_payload <= (others=>'0');
                      EA <= S0;
                   else
                      EA <= S2;
                   end if;
             end case;
          end if;
       end if;
   end process;

   func_xor(0) <=  data_in(1) xor data_in(0);
   func_xor(1) <=  data_in(2) xor data_in(1);
   func_xor(2) <=  data_in(3) xor data_in(2);
   func_xor(3) <=  data_in(4) xor data_in(3);
   func_xor(4) <=  data_in(5) xor data_in(4);
   func_xor(5) <=  data_in(6) xor data_in(5);
   func_xor(6) <=  data_in(7) xor data_in(6);

   data_out(0) <= data_in (0) when EA = S0 or EA = S1
      else func_xor(0);
   data_out(1) <= data_in (1) when EA = S0 or EA = S1
      else func_xor(1);
   data_out(2) <= data_in (2) when EA = S0 or EA = S1
      else func_xor(2);
   data_out(3) <= data_in (3) when EA = S0 or EA = S1
      else func_xor(3);
   data_out(4) <= data_in (4) when EA = S0 or EA = S1
      else func_xor(4);
   data_out(5) <= data_in (5) when EA = S0 or EA = S1
      else func_xor(5);
   data_out(6) <= data_in (6) when EA = S0 or EA = S1
      else func_xor(6);
   data_out(7) <= data_in (7);

   tx <= rx;
   clock_tx_out <= clock_tx_in;
   credit_i_out <= credit_i_in;

end encoder_gray;
```

Tabela A.2: Código VHDL do decodificador *Gray* com largura de *flit* igual a 8 bits.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity decoder_gray is
port(
   clock_rx_in : in std_logic;
   reset : in std_logic;
   data_in : in std_logic_vector(7 downto 0);
   rx : in std_logic;
   clock_rx_out : out std_logic;
   data_out : out std_logic_vector(7 downto 0);
   tx : out std_logic;
   credit_o_in : in std_logic;
```

```
   credit_o_out : out std_logic
);
end decoder_gray;

architecture decoder_gray of decoder_gray is

--tamanho do pacote; segundo flit do cabecalho
signal tamanho_pacote : std_logic_vector(7 downto 0);
signal cont_flit_payload : std_logic_vector(7 downto 0);
type fsm is (S0,S1,s2);
signal EA : fsm;
signal func_xor : std_logic_vector(6 downto 0);

begin
   process(reset, clock_rx_in)
   begin
      if reset='1' then
         EA <= S0;
         cont_flit_payload <= (others=>'0');
      elsif clock_rx_in'event and clock_rx_in='1' then
         if credit_o_in='1' then
            case EA is
               -- estado inicial, antes de uma transmissao
               when S0 =>
                  if rx ='1' then
                     EA <= S1;
                  else
                     EA <= S0;
                  end if;
               -- inicio de transmissao, transmitindo cabecalho
               when S1 =>
                  if rx ='1' then
                     tamanho_pacote <= data_in;
                     cont_flit_payload <= cont_flit_payload + 1;
                     EA <= S2;
                  else
                     EA <= S1;
                  end if;
               when S2 =>
                  cont_flit_payload <= cont_flit_payload + 1;
                  if cont_flit_payload = tamanho_pacote then
                     cont_flit_payload <= (others=>'0');
                     EA <= S0;
                  else
                     EA <= S2;
                  end if;
            end case;
         end if;
      end if;
   end process;

   func_xor(0) <=  data_in(0) xor func_xor(1);
   func_xor(1) <=  data_in(1) xor func_xor(2);
   func_xor(2) <=  data_in(2) xor func_xor(3);
   func_xor(3) <=  data_in(3) xor func_xor(4);
   func_xor(4) <=  data_in(4) xor func_xor(5);
   func_xor(5) <=  data_in(5) xor func_xor(6);
   func_xor(6) <=  data_in(6) xor data_in(7);

   data_out(0) <= data_in (0) when EA = S0 or EA = S1
      else func_xor(0);
   data_out(1) <= data_in (1) when EA = S0 or EA = S1
      else func_xor(1);
   data_out(2) <= data_in (2) when EA = S0 or EA = S1
      else func_xor(2);
   data_out(3) <= data_in (3) when EA = S0 or EA = S1
      else func_xor(3);
   data_out(4) <= data_in (4) when EA = S0 or EA = S1
      else func_xor(4);
```

```
   data_out(5) <= data_in (5) when EA = S0 or EA = S1
      else func_xor(5);
   data_out(6) <= data_in (6) when EA = S0 or EA = S1
      else func_xor(6);
   data_out(7) <= data_in (7);

   tx <= rx;
   clock_rx_out <= clock_rx_in;
   credit_o_out <= credit_o_in;

end decoder_gray;
```

Tabela A.3: Código VHDL do codificador *Transition* com largura de *flit* igual a 8 bits.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity encoder_transition is
port(
   clock_tx_in : in std_logic;
   reset : in std_logic;
   data_in : in std_logic_vector(7 downto 0);
   rx : in std_logic;
   clock_tx_out : out std_logic;
   data_out : out std_logic_vector(7 downto 0);
   tx : out std_logic;
   credit_i_in : in std_logic;
   credit_i_out : out std_logic
);
end encoder_transition;

architecture encoder_transition of encoder_transition is

--tamanho do pacote; segundo flit do cabecalho
signal tamanho_pacote : std_logic_vector(7 downto 0);
signal  cont flit payload,  data ant,  data to out  :  std logic vector(7
downto 0);
type fsm is (S0,S1,s2);
signal EA : fsm;
signal func_xor : std_logic_vector(7 downto 0);

begin
   process(reset, clock_tx_in)
   begin
      if reset='1' then
         data_ant <= (others=>'0');
         EA <= S0;
         cont_flit_payload <= (others=>'0');
      elsif clock_tx_in'event and clock_tx_in='0' then
         if credit_i_in='1' then
            case EA is
               -- estado inicial, antes de uma transmissao
               when S0 =>
                  if rx ='1' then
                     EA <= S1;
                     data_ant <= data_in;
                  else
                     EA <= S0;
                  end if;
               -- inicio de transmissao, transmitindo cabecalho
               when S1 =>
                  if rx ='1' then
                     tamanho_pacote <= data_in;
                     cont_flit_payload <= cont_flit_payload + 1;
                     data_ant <= data_in;
                     EA <= S2;
                  else
```

```
                    EA <= S1;
                 end if;
              when S2 =>
                 data_ant <= data_in;
                 cont_flit_payload <= cont_flit_payload + 1;
                 if cont_flit_payload = tamanho_pacote then
                    cont_flit_payload <= (others=>'0');
                    EA <= S0;
                 else
                    EA <= S2;
                 end if;
              end case;
           end if;
        end if;
     end process;

   func_xor(0) <=  data_in(0) xor data_ant(0);
   func_xor(1) <=  data_in(1) xor data_ant(1);
   func_xor(2) <=  data_in(2) xor data_ant(2);
   func_xor(3) <=  data_in(3) xor data_ant(3);
   func_xor(4) <=  data_in(4) xor data_ant(4);
   func_xor(5) <=  data_in(5) xor data_ant(5);
   func_xor(6) <=  data_in(6) xor data_ant(6);
   func_xor(7) <=  data_in(7) xor data_ant(7);

   data_out <= data_in when EA = S0 or EA = S1
      else func_xor;

   tx <= rx;
   clock_tx_out <= clock_tx_in;
   credit_i_out <= credit_i_in;

end encoder_transition;
```

Tabela A.4: Código VHDL do decodificador *Transition* com largura de *flit* igual a 8 bits.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity decoder_transition is
port(
   clock_rx_in : in std_logic;
   reset : in std_logic;
   data_in : in std_logic_vector(7 downto 0);
   rx : in std_logic;
   clock_rx_out : out std_logic;
   data_out : out std_logic_vector(7 downto 0);
   tx : out std_logic;
   credit_o_in : in std_logic;
   credit_o_out : out std_logic
);
end decoder_transition;

architecture decoder_transition of decoder_transition is

--tamanho do pacote; segundo flit do cabecalho
signal tamanho_pacote : std_logic_vector(7 downto 0);
signal  cont flit payload,  data ant,  data to out  :  std logic vector(7
downto 0);
type fsm is (S0,S1,s2);
signal EA : fsm;
signal func_xor : std_logic_vector(7 downto 0);

begin
   process(reset, clock_rx_in)
   begin
```

```
      if reset='1' then
         data_ant <= (others=>'0');
         EA <= S0;
         cont_flit_payload <= (others=>'0');
      elsif clock_rx_in'event and clock_rx_in='1' then
         if credit_o_in='1' then
            case EA is
               -- estado inicial, antes de uma transmissao
               when S0 =>
                  if rx ='1' then
                     data_ant <= data_in;
                     EA <= S1;
                  else
                     EA <= S0;
                  end if;
               -- inicio de transmissao, transmitindo cabecalho
               when S1 =>
                  if rx ='1' then
                     tamanho_pacote <= data_in;
                     data_ant <= data_in;
                     cont_flit_payload <= cont_flit_payload + 1;
                     EA <= S2;
                  else
                     EA <= S1;
                  end if;
               when S2 =>
                  data_ant <= data_to_out;
                  cont_flit_payload <= cont_flit_payload + 1;
                  if cont_flit_payload = tamanho_pacote then
                     cont_flit_payload <= (others=>'0');
                     EA <= S0;
                  else
                     EA <= S2;
                  end if;
            end case;
         end if;
      end if;
   end process;

   func_xor(0) <=  data_in(0) xor data_ant(0);
   func_xor(1) <=  data_in(1) xor data_ant(1);
   func_xor(2) <=  data_in(2) xor data_ant(2);
   func_xor(3) <=  data_in(3) xor data_ant(3);
   func_xor(4) <=  data_in(4) xor data_ant(4);
   func_xor(5) <=  data_in(5) xor data_ant(5);
   func_xor(6) <=  data_in(6) xor data_ant(6);
   func_xor(7) <=  data_in(7) xor data_ant(7);

   data_to_out <= data_in when EA = S0 or EA = S1
      else func_xor;

   data_out <= data_to_out;
   tx <= rx;
   clock_rx_out <= clock_rx_in;
   credit_o_out <= credit_o_in;

end decoder_transition;
```

Tabela A.5: Código VHDL do codificador *Adaptive Encoding* com largura de *flit* igual a 8 bits.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
```

```
entity encoderAdaptive is
port(
   clock_tx_in : in std_logic;
   reset : in std_logic;
   data_in : in std_logic_vector(7 downto 0);
   rx : in std_logic;
   clock_tx_out : out std_logic;
   data_out : out std_logic_vector(7 downto 0);
   tx : out std_logic;
   credit_i_in : in std_logic;
   credit_i_out : out std_logic
);
end encoderAdaptive;

architecture encoder of encoderAdaptive is

type fsm is (S0,S1,S0B);
signal EA : fsm;

-- contador geral - janela de 64 bits
signal wincnt : std_logic_vector(5 downto 0);
-- contadores de transicao max 32
--subtype regcont is  std_logic_vector(5 downto 0);
--type cont is array(7 downto 0) of regcont;
signal n00_0,n11_0,nT_0 : std_logic_vector(5 downto 0);
signal n00_1,n11_1,nT_1 : std_logic_vector(5 downto 0);
signal n00_2,n11_2,nT_2 : std_logic_vector(5 downto 0);
signal n00_3,n11_3,nT_3 : std_logic_vector(5 downto 0);
signal n00_4,n11_4,nT_4 : std_logic_vector(5 downto 0);
signal n00_5,n11_5,nT_5 : std_logic_vector(5 downto 0);
signal n00_6,n11_6,nT_6 : std_logic_vector(5 downto 0);
signal n00_7,n11_7,nT_7 : std_logic_vector(5 downto 0);
signal data_ant : std_logic_vector(7 downto 0);
signal tamanho_pacote : std_logic_vector(7 downto 0);
signal cont_flit_payload : std_logic_vector(7 downto 0);

--sinais da logica combinacional para codificacao
signal  cond 1,cond 2,cond 3,cond 4,cond 5  :  std logic vector(7  downto
0);
signal func3 : std_logic_vector(7 downto 0);

begin
   process(reset, clock_tx_in)
   begin
      if reset='1' then
         EA <= S0;
         wincnt <= (others=>'0');
         cont_flit_payload <= (others=>'0');
         n00_0 <= (others=>'0');
         n11_0 <= (others=>'0');
         nT_0 <= (others=>'0');
         n00_1 <= (others=>'0');
         n11_1 <= (others=>'0');
         nT_1 <= (others=>'0');
         n00_2 <= (others=>'0');
         n11_2 <= (others=>'0');
         nT_2 <= (others=>'0');
         n00_3 <= (others=>'0');
         n11_3 <= (others=>'0');
         nT_3 <= (others=>'0');
         n00_4 <= (others=>'0');
         n11_4 <= (others=>'0');
         nT_4 <= (others=>'0');
         n00_5 <= (others=>'0');
         n11_5 <= (others=>'0');
         nT_5 <= (others=>'0');
         n00_6 <= (others=>'0');
         n11_6 <= (others=>'0');
```

```
         nT_6 <= (others=>'0');
         n00_7 <= (others=>'0');
         n11_7 <= (others=>'0');
         nT_7 <= (others=>'0');

      elsif clock_tx_in'event and clock_tx_in='0' then
         if credit_i_in='1' then
            case EA is
               -- estado inicial, antes de uma transmissao
               when S0 =>
                  if rx ='1' then
                     wincnt <= wincnt + 1;
                     data_ant <= data_in;
                     EA <= S1;
                  else
                     EA <= S0;
                  end if;
               when S0B =>
                  if rx ='1' then
                     wincnt <= wincnt + 1;
                     data_ant <= data_in;
                     cont_flit_payload <= cont_flit_payload + 1;
                     EA <= S1;
                  else
                     EA <= S0B;
                  end if;
               -- transmissao
               when S1 =>
                  wincnt <= wincnt + 1;
                  if wincnt=1 and cont_flit_payload=0 then
                     tamanho_pacote <= data_in;
                  end if;
                  if wincnt>0 then
                     cont_flit_payload <= cont_flit_payload + 1;
                  end if;

                  if data_in(0) = data_ant(0) then
                     if data_in(0) = '1' then
                        n11_0 <= n11_0 + 1;
                     elsif data_in(0) = '0' then
                        n00_0 <= n00_0 + 1;
                     end if;
                  elsif data_in(0) = '0' and data_ant(0) = '1' then
                     nT_0 <= nT_0 + 1;
                  end if;

                  if data_in(1) = data_ant(1) then
                     if data_in(1) = '1' then
                        n11_1 <= n11_1 + 1;
                     elsif data_in(1) = '0' then
                        n00_1 <= n00_1 + 1;
                     end if;
                  elsif data_in(1) = '0' and data_ant(1) = '1' then
                     nT_1 <= nT_1 + 1;
                  end if;

                  if data_in(2) = data_ant(2) then
                     if data_in(2) = '1' then
                        n11_2 <= n11_2 + 1;
                     elsif data_in(2) = '0' then
                        n00_2 <= n00_2 + 1;
                     end if;
                  elsif data_in(2) = '0' and data_ant(2) = '1' then
                     nT_2 <= nT_2 + 1;
                  end if;

                  if data_in(3) = data_ant(3) then
                     if data_in(3) = '1' then
                        n11_3 <= n11_3 + 1;
```

```
                        elsif data_in(3) = '0' then
                           n00_3 <= n00_3 + 1;
                        end if;
                     elsif data_in(3) = '0' and data_ant(3) = '1' then
                        nT_3 <= nT_3 + 1;
                     end if;

                     if data_in(4) = data_ant(4) then
                        if data_in(4) = '1' then
                           n11_4 <= n11_4 + 1;
                        elsif data_in(4) = '0' then
                           n00_4 <= n00_4 + 1;
                        end if;
                     elsif data_in(4) = '0' and data_ant(4) = '1' then
                        nT_4 <= nT_4 + 1;
                     end if;

                     if data_in(5) = data_ant(5) then
                        if data_in(5) = '1' then
                           n11_5 <= n11_5 + 1;
                        elsif data_in(5) = '0' then
                           n00_5 <= n00_5 + 1;
                        end if;
                     elsif data_in(5) = '0' and data_ant(5) = '1' then
                        nT_5 <= nT_5 + 1;
                     end if;

                     if data_in(6) = data_ant(6) then
                        if data_in(6) = '1' then
                           n11_6 <= n11_6 + 1;
                        elsif data_in(6) = '0' then
                           n00_6 <= n00_6 + 1;
                        end if;
                     elsif data_in(6) = '0' and data_ant(6) = '1' then
                        nT_6 <= nT_6 + 1;
                     end if;

                     if data_in(7) = data_ant(7) then
                        if data_in(7) = '1' then
                           n11_7 <= n11_7 + 1;
                        elsif data_in(7) = '0' then
                           n00_7 <= n00_7 + 1;
                        end if;
                     elsif data_in(7) = '0' and data_ant(7) = '1' then
                        nT_7 <= nT_7 + 1;
                     end if;

                     data_ant <= data_in;
                     --condicao para resetar contadores
                     if wincnt = 63 or cont_flit_payload = tamanho_pacote
then
                        --EA <= S0;
                        wincnt <= (others=>'0');
                        n00_0 <= (others=>'0');
                        n11_0 <= (others=>'0');
                        nT_0 <= (others=>'0');
                        n00_1 <= (others=>'0');
                        n11_1 <= (others=>'0');
                        nT_1 <= (others=>'0');
                        n00_2 <= (others=>'0');
                        n11_2 <= (others=>'0');
                        nT_2 <= (others=>'0');
                        n00_3 <= (others=>'0');
                        n11_3 <= (others=>'0');
                        nT_3 <= (others=>'0');
                        n00_4 <= (others=>'0');
                        n11_4 <= (others=>'0');
                        nT_4 <= (others=>'0');
                        n00_5 <= (others=>'0');
```

```
                    n11_5 <= (others=>'0');
                    nT_5 <= (others=>'0');
                    n00_6 <= (others=>'0');
                    n11_6 <= (others=>'0');
                    nT_6 <= (others=>'0');
                    n00_7 <= (others=>'0');
                    n11_7 <= (others=>'0');
                    nT_7 <= (others=>'0');
                 end if;
                 if cont_flit_payload = tamanho_pacote then
                    cont_flit_payload <= (others=>'0');
                    EA <= S0;
                 elsif wincnt = 63 then
                    EA <= S0B;
                 end if;
           end case;
        end if;
     end if;
  end process;

  --n11>nT
  cond_1(0) <= '1'when n11_0>nt_0 else '0';
  --n00>nT
  cond_2(0) <= '1'when n00_0>nt_0 else '0';
  --nT>n00
  cond_3(0) <= '1'when nT_0>n00_0 else '0';
  --nT>n11
  cond_4(0) <= '1'when nT_0>n11_0 else '0';
  --nT>=n00
  cond_5(0) <= '1'when nT_0>=n00_0 else '0';

  cond_1(1) <= '1'when n11_1>nt_1 else '0';
  cond_2(1) <= '1'when n00_1>nt_1 else '0';
  cond_3(1) <= '1'when nT_1>n00_1 else '0';
  cond_4(1) <= '1'when nT_1>n11_1 else '0';
  cond_5(1) <= '1'when nT_1>=n00_1 else '0';

  cond_1(2) <= '1'when n11_2>nt_2 else '0';
  cond_2(2) <= '1'when n00_2>nt_2 else '0';
  cond_3(2) <= '1'when nT_2>n00_2 else '0';
  cond_4(2) <= '1'when nT_2>n11_2 else '0';
  cond_5(2) <= '1'when nT_2>=n00_2 else '0';

  cond_1(3) <= '1'when n11_3>nt_3 else '0';
  cond_2(3) <= '1'when n00_3>nt_3 else '0';
  cond_3(3) <= '1'when nT_3>n00_3 else '0';
  cond_4(3) <= '1'when nT_3>n11_3 else '0';
  cond_5(3) <= '1'when nT_3>=n00_3 else '0';

  cond_1(4) <= '1'when n11_4>nt_4 else '0';
  cond_2(4) <= '1'when n00_4>nt_4 else '0';
  cond_3(4) <= '1'when nT_4>n00_4 else '0';
  cond_4(4) <= '1'when nT_4>n11_4 else '0';
  cond_5(4) <= '1'when nT_4>=n00_4 else '0';

  cond_1(5) <= '1'when n11_5>nt_5 else '0';
  cond_2(5) <= '1'when n00_5>nt_5 else '0';
  cond_3(5) <= '1'when nT_5>n00_5 else '0';
  cond_4(5) <= '1'when nT_5>n11_5 else '0';
  cond_5(5) <= '1'when nT_5>=n00_5 else '0';

  cond_1(6) <= '1'when n11_6>nt_6 else '0';
  cond_2(6) <= '1'when n00_6>nt_6 else '0';
  cond_3(6) <= '1'when nT_6>n00_6 else '0';
  cond_4(6) <= '1'when nT_6>n11_6 else '0';
  cond_5(6) <= '1'when nT_6>=n00_6 else '0';

  cond_1(7) <= '1'when n11_7>nt_7 else '0';
  cond_2(7) <= '1'when n00_7>nt_7 else '0';
```

```
   cond_3(7) <= '1'when nT_7>n00_7 else '0';
   cond_4(7) <= '1'when nT_7>n11_7 else '0';
   cond_5(7) <= '1'when nT_7>=n00_7 else '0';

   func3(0) <=  data_in(0) xor data_ant(0);
   func3(1) <=  data_in(1) xor data_ant(1);
   func3(2) <=  data_in(2) xor data_ant(2);
   func3(3) <=  data_in(3) xor data_ant(3);
   func3(4) <=  data_in(4) xor data_ant(4);
   func3(5) <=  data_in(5) xor data_ant(5);
   func3(6) <=  data_in(6) xor data_ant(6);
   func3(7) <=  data_in(7) xor data_ant(7);

   data_out(0) <= func3(0) when cond_1(0) = '1' and cond_2(0) = '1'
      else not func3(0) when cond_3(0) = '1' and cond_4(0) = '1'
      else not data_in(0) when cond_1(0) = '1' and cond_5(0) = '1'
      else data_in(0);

   data_out(1) <= func3(1) when cond_1(1) = '1' and cond_2(1) = '1'
      else not func3(1) when cond_3(1) = '1' and cond_4(1) = '1'
      else not data_in(1) when cond_1(1) = '1' and cond_5(1) = '1'
      else data_in(1);

   data_out(2) <= func3(2) when cond_1(2) = '1' and cond_2(2) = '1'
      else not func3(2) when cond_3(2) = '1' and cond_4(2) = '1'
      else not data_in(2) when cond_1(2) = '1' and cond_5(2) = '1'
      else data_in(2);
   data_out(3) <= func3(3) when cond_1(3) = '1' and cond_2(3) = '1'
      else not func3(3) when cond_3(3) = '1' and cond_4(3) = '1'
      else not data_in(3) when cond_1(3) = '1' and cond_5(3) = '1'
      else data_in(3);

   data_out(4) <= func3(4) when cond_1(4) = '1' and cond_2(4) = '1'
      else not func3(4) when cond_3(4) = '1' and cond_4(4) = '1'
      else not data_in(4) when cond_1(4) = '1' and cond_5(4) = '1'
      else data_in(4);

   data_out(5) <= func3(5) when cond_1(5) = '1' and cond_2(5) = '1'
      else not func3(5) when cond_3(5) = '1' and cond_4(5) = '1'
      else not data_in(5) when cond_1(5) = '1' and cond_5(5) = '1'
      else data_in(5);

   data_out(6) <= func3(6) when cond_1(6) = '1' and cond_2(6) = '1'
      else not func3(6) when cond_3(6) = '1' and cond_4(6) = '1'
      else not data_in(6) when cond_1(6) = '1' and cond_5(6) = '1'
      else data_in(6);

   data_out(7) <= func3(7) when cond_1(7) = '1' and cond_2(7) = '1'
      else not func3(7) when cond_3(7) = '1' and cond_4(7) = '1'
      else not data_in(7) when cond_1(7) = '1' and cond_5(7) = '1'
      else data_in(7);

   tx <= rx;
   clock_tx_out <= clock_tx_in;
   credit_i_out <= credit_i_in;

end encoder;
```

Tabela A.6: Código VHDL do decodificador *Adaptive Encoding* com largura de *flit* igual a 8 bits.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity decoder is
port(
   clock_rx_in : in std_logic;
```

```
   reset : in std_logic;
   data_in : in std_logic_vector(7 downto 0);
   rx : in std_logic;
   clock_rx_out : out std_logic;
   data_out : out std_logic_vector(7 downto 0);
   tx : out std_logic;
   credit_o_in : in std_logic;
   credit_o_out : out std_logic
);
end decoder;

architecture decoder of decoder is

type fsm is (S0,S1,S0B);
signal EA : fsm;

-- contador geral - janela de 64 bits
signal wincnt : std_logic_vector(5 downto 0);
-- contadores de transicao max 32
signal n00_0,n11_0,nT_0 : std_logic_vector(5 downto 0);
signal n00_1,n11_1,nT_1 : std_logic_vector(5 downto 0);
signal n00_2,n11_2,nT_2 : std_logic_vector(5 downto 0);
signal n00_3,n11_3,nT_3 : std_logic_vector(5 downto 0);
signal n00_4,n11_4,nT_4 : std_logic_vector(5 downto 0);
signal n00_5,n11_5,nT_5 : std_logic_vector(5 downto 0);
signal n00_6,n11_6,nT_6 : std_logic_vector(5 downto 0);
signal n00_7,n11_7,nT_7 : std_logic_vector(5 downto 0);

signal data_ant,data_to_out : std_logic_vector(7 downto 0);
signal tamanho_pacote : std_logic_vector(7 downto 0);
signal cont_flit_payload : std_logic_vector(7 downto 0);

signal  cond 1,cond 2,cond 3,cond 4,cond 5  :  std logic vector(7  downto
0);
signal func3 : std_logic_vector(7 downto 0);

begin
   process(reset, clock_rx_in)
   begin
      if reset='1' then
         EA <= S0;
         wincnt <= (others=>'0');
         cont_flit_payload <= (others=>'0');
         n00_0 <= (others=>'0');
         n11_0 <= (others=>'0');
         nT_0 <= (others=>'0');
         n00_1 <= (others=>'0');
         n11_1 <= (others=>'0');
         nT_1 <= (others=>'0');
         n00_2 <= (others=>'0');
         n11_2 <= (others=>'0');
         nT_2 <= (others=>'0');
         n00_3 <= (others=>'0');
         n11_3 <= (others=>'0');
         nT_3 <= (others=>'0');
         n00_4 <= (others=>'0');
         n11_4 <= (others=>'0');
         nT_4 <= (others=>'0');
         n00_5 <= (others=>'0');
         n11_5 <= (others=>'0');
         nT_5 <= (others=>'0');
         n00_6 <= (others=>'0');
         n11_6 <= (others=>'0');
         nT_6 <= (others=>'0');
         n00_7 <= (others=>'0');
         n11_7 <= (others=>'0');
         nT_7 <= (others=>'0');
      elsif clock_rx_in'event and clock_rx_in='1' then
         if credit_o_in='1' then
```

```
            case EA is
               -- estado inicial, antes de uma transmissao
               when S0 =>
                  if rx ='1' then
                     wincnt <= wincnt + 1;
                     data_ant <= data_to_out;
                     EA <= S1;
                  else
                     EA <= S0;
                  end if;
               when S0B =>
                  if rx ='1' then
                     wincnt <= wincnt + 1;
                     data_ant <= data_in;
                     cont_flit_payload <= cont_flit_payload + 1;
                     EA <= S1;
                  else
                     EA <= S0B;
                  end if;
               -- transmissao
               when S1 =>
                  wincnt <= wincnt + 1;
                  if wincnt=1 and cont_flit_payload=0 then
                     tamanho_pacote <= data_in;
                  end if;
                  if wincnt>0 then
                     cont_flit_payload <= cont_flit_payload + 1;
                  end if;

                  if data_to_out(0) = data_ant(0) then
                     if data_to_out(0) = '1' then
                        n11_0 <= n11_0 + 1;
                     elsif data_to_out(0) = '0' then
                        n00_0 <= n00_0 + 1;
                     end if;
                  elsif  data to out(0)  =  '0'  and  data ant(0)  =  '1'
then
                     nT_0 <= nT_0 + 1;
                  end if;

                  if data_to_out(1) = data_ant(1) then
                     if data_to_out(1) = '1' then
                        n11_1 <= n11_1 + 1;
                     elsif data_to_out(1) = '0' then
                        n00_1 <= n00_1 + 1;
                     end if;
                  elsif  data to out(1)  =  '0'  and  data ant(1)  =  '1'
then
                     nT_1 <= nT_1 + 1;
                  end if;

                  if data_to_out(2) = data_ant(2) then
                     if data_to_out(2) = '1' then
                        n11_2 <= n11_2 + 1;
                     elsif data_to_out(2) = '0' then
                        n00_2 <= n00_2 + 1;
                     end if;
                  elsif  data to out(2)  =  '0'  and  data ant(2)  =  '1'
then
                     nT_2 <= nT_2 + 1;
                  end if;

                  if data_to_out(3) = data_ant(3) then
                     if data_to_out(3) = '1' then
                        n11_3 <= n11_3 + 1;
                     elsif data_to_out(3) = '0' then
                        n00_3 <= n00_3 + 1;
                     end if;
                  elsif  data to out(3)  =  '0'  and  data ant(3)  =  '1'
```

```
then
                    nT_3 <= nT_3 + 1;
                end if;

                if data_to_out(4) = data_ant(4) then
                    if data_to_out(4) = '1' then
                        n11_4 <= n11_4 + 1;
                    elsif data_to_out(4) = '0' then
                        n00_4 <= n00_4 + 1;
                    end if;
                elsif data to out(4) = '0' and data ant(4) = '1'
then
                    nT_4 <= nT_4 + 1;
                end if;

                if data_to_out(5) = data_ant(5) then
                    if data_to_out(5) = '1' then
                        n11_5 <= n11_5 + 1;
                    elsif data_to_out(5) = '0' then
                        n00_5 <= n00_5 + 1;
                    end if;
                elsif data_to_out(5) = '0' and data ant(5) = '1'
then
                    nT_5 <= nT_5 + 1;
                end if;

                if data_to_out(6) = data_ant(6) then
                    if data_to_out(6) = '1' then
                        n11_6 <= n11_6 + 1;
                    elsif data_to_out(6) = '0' then
                        n00_6 <= n00_6 + 1;
                    end if;
                elsif data to out(6) = '0' and data ant(6) = '1'
then
                    nT_6 <= nT_6 + 1;
                end if;

                if data_to_out(7) = data_ant(7) then
                    if data_to_out(7) = '1' then
                        n11_7 <= n11_7 + 1;
                    elsif data_to_out(7) = '0' then
                        n00_7 <= n00_7 + 1;
                    end if;
                elsif data to out(7) = '0' and data ant(7) = '1'
then
                    nT_7 <= nT_7 + 1;
                end if;

                data_ant <= data_to_out;
                --condicao para resetar contadores
                if wincnt = 63 or cont flit payload = tamanho pacote
then
                    wincnt <= (others=>'0');
                    n00_0 <= (others=>'0');
                    n11_0 <= (others=>'0');
                    nT_0 <= (others=>'0');
                    n00_1 <= (others=>'0');
                    n11_1 <= (others=>'0');
                    nT_1 <= (others=>'0');
                    n00_2 <= (others=>'0');
                    n11_2 <= (others=>'0');
                    nT_2 <= (others=>'0');
                    n00_3 <= (others=>'0');
                    n11_3 <= (others=>'0');
                    nT_3 <= (others=>'0');
                    n00_4 <= (others=>'0');
                    n11_4 <= (others=>'0');
                    nT_4 <= (others=>'0');
                    n00_5 <= (others=>'0');
```

```
                    n11_5 <= (others=>'0');
                    nT_5 <= (others=>'0');
                    n00_6 <= (others=>'0');
                    n11_6 <= (others=>'0');
                    nT_6 <= (others=>'0');
                    n00_7 <= (others=>'0');
                    n11_7 <= (others=>'0');
                    nT_7 <= (others=>'0');
                end if;

                if cont_flit_payload = tamanho_pacote then
                    cont_flit_payload <= (others=>'0');
                    EA <= S0;
                elsif wincnt = 63 then
                    EA <= S0B;
                end if;
            end case;
        end if;
    end if;
end process;

--n11>nT
cond_1(0) <= '1'when n11_0>nt_0 else '0';
--n00>nT
cond_2(0) <= '1'when n00_0>nt_0 else '0';
--nT>n00
cond_3(0) <= '1'when nT_0>n00_0 else '0';
--nT>n11
cond_4(0) <= '1'when nT_0>n11_0 else '0';
--nT>=n00
cond_5(0) <= '1'when nT_0>=n00_0 else '0';

cond_1(1) <= '1'when n11_1>nt_1 else '0';
cond_2(1) <= '1'when n00_1>nt_1 else '0';
cond_3(1) <= '1'when nT_1>n00_1 else '0';
cond_4(1) <= '1'when nT_1>n11_1 else '0';
cond_5(1) <= '1'when nT_1>=n00_1 else '0';

cond_1(2) <= '1'when n11_2>nt_2 else '0';
cond_2(2) <= '1'when n00_2>nt_2 else '0';
cond_3(2) <= '1'when nT_2>n00_2 else '0';
cond_4(2) <= '1'when nT_2>n11_2 else '0';
cond_5(2) <= '1'when nT_2>=n00_2 else '0';

cond_1(3) <= '1'when n11_3>nt_3 else '0';
cond_2(3) <= '1'when n00_3>nt_3 else '0';
cond_3(3) <= '1'when nT_3>n00_3 else '0';
cond_4(3) <= '1'when nT_3>n11_3 else '0';
cond_5(3) <= '1'when nT_3>=n00_3 else '0';

cond_1(4) <= '1'when n11_4>nt_4 else '0';
cond_2(4) <= '1'when n00_4>nt_4 else '0';
cond_3(4) <= '1'when nT_4>n00_4 else '0';
cond_4(4) <= '1'when nT_4>n11_4 else '0';
cond_5(4) <= '1'when nT_4>=n00_4 else '0';

cond_1(5) <= '1'when n11_5>nt_5 else '0';
cond_2(5) <= '1'when n00_5>nt_5 else '0';
cond_3(5) <= '1'when nT_5>n00_5 else '0';
cond_4(5) <= '1'when nT_5>n11_5 else '0';
cond_5(5) <= '1'when nT_5>=n00_5 else '0';

cond_1(6) <= '1'when n11_6>nt_6 else '0';
cond_2(6) <= '1'when n00_6>nt_6 else '0';
cond_3(6) <= '1'when nT_6>n00_6 else '0';
cond_4(6) <= '1'when nT_6>n11_6 else '0';
cond_5(6) <= '1'when nT_6>=n00_6 else '0';

cond_1(7) <= '1'when n11_7>nt_7 else '0';
```

```
    cond_2(7) <= '1'when n00_7>nt_7 else '0';
    cond_3(7) <= '1'when nT_7>n00_7 else '0';
    cond_4(7) <= '1'when nT_7>n11_7 else '0';
    cond_5(7) <= '1'when nT_7>=n00_7 else '0';

    func3(0) <=  data_in(0) xor data_ant(0);
    func3(1) <=  data_in(1) xor data_ant(1);
    func3(2) <=  data_in(2) xor data_ant(2);
    func3(3) <=  data_in(3) xor data_ant(3);
    func3(4) <=  data_in(4) xor data_ant(4);
    func3(5) <=  data_in(5) xor data_ant(5);
    func3(6) <=  data_in(6) xor data_ant(6);
    func3(7) <=  data_in(7) xor data_ant(7);

    data_to_out(0) <= func3(0) when cond_1(0) = '1' and cond_2(0) = '1'
       else not func3(0) when cond_3(0) = '1' and cond_4(0) = '1'
       else not data_in(0) when cond_1(0) = '1' and cond_5(0) = '1'
       else data_in(0);

    data_to_out(1) <= func3(1) when cond_1(1) = '1' and cond_2(1) = '1'
       else not func3(1) when cond_3(1) = '1' and cond_4(1) = '1'
       else not data_in(1) when cond_1(1) = '1' and cond_5(1) = '1'
       else data_in(1);

    data_to_out(2) <= func3(2) when cond_1(2) = '1' and cond_2(2) = '1'
       else not func3(2) when cond_3(2) = '1' and cond_4(2) = '1'
       else not data_in(2) when cond_1(2) = '1' and cond_5(2) = '1'
       else data_in(2);

    data_to_out(3) <= func3(3) when cond_1(3) = '1' and cond_2(3) = '1'
       else not func3(3) when cond_3(3) = '1' and cond_4(3) = '1'
       else not data_in(3) when cond_1(3) = '1' and cond_5(3) = '1'
       else data_in(3);

    data_to_out(4) <= func3(4) when cond_1(4) = '1' and cond_2(4) = '1'
       else not func3(4) when cond_3(4) = '1' and cond_4(4) = '1'
       else not data_in(4) when cond_1(4) = '1' and cond_5(4) = '1'
       else data_in(4);

    data_to_out(5) <= func3(5) when cond_1(5) = '1' and cond_2(5) = '1'
       else not func3(5) when cond_3(5) = '1' and cond_4(5) = '1'
       else not data_in(5) when cond_1(5) = '1' and cond_5(5) = '1'
       else data_in(5);

    data_to_out(6) <= func3(6) when cond_1(6) = '1' and cond_2(6) = '1'
       else not func3(6) when cond_3(6) = '1' and cond_4(6) = '1'
       else not data_in(6) when cond_1(6) = '1' and cond_5(6) = '1'
       else data_in(6);

    data_to_out(7) <= func3(7) when cond_1(7) = '1' and cond_2(7) = '1'
       else not func3(7) when cond_3(7) = '1' and cond_4(7) = '1'
       else not data_in(7) when cond_1(7) = '1' and cond_5(7) = '1'
       else data_in(7);

    data_out <= data_to_out;
    tx <= rx;
    clock_rx_out <= clock_rx_in;
    credit_o_out <= credit_o_in;

end decoder;
```

Tabela A.7: Código VHDL do codificador *Bus-Invert* com 1 *cluster* e largura de *flit* igual a 8 bits.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
```

```
entity BusInv_encoder2 is
port(
   clock_tx_in : in std_logic;
   reset : in std_logic;
   data_in : in std_logic_vector(7 downto 0);
   rx : in std_logic;
   clock_tx_out : out std_logic;
   data_out : out std_logic_vector(8 downto 0); -- 9 bits, 8 data + 1
inv
   tx : out std_logic;
   credit_i_in : in std_logic;
   credit_i_out : out std_logic
);
end BusInv_encoder2;

architecture BusInv_encoder2 of BusInv_encoder2 is

signal data_ant : std_logic_vector(8 downto 0);
signal dataIn_xor_dataAnt : std_logic_vector(8 downto 0);
signal majority : std_logic;

begin

   s: entity work.majority_9
   port map(
      vet => dataIn_xor_dataAnt,
      s4A => majority
   );

   process(reset, clock_tx_in)
   begin
      if reset='1' then
         data_out(8 downto 0) <= (others=>'0');
      elsif clock_tx_in'event and clock_tx_in='1' then
         if credit_i_in='1' then
            if majority='0' then
               data_out(7 downto 0) <=data_in;
               data_out(8) <= '0';
            else
               data_out(7 downto 0) <=not data_in;
               data_out(8) <= '1';
            end if;
         end if;
      end if;
   end process;

   process(reset, clock_tx_in)
   begin
      if reset='1' then
         data_ant <= (others=>'0');
      elsif clock_tx_in'event and clock_tx_in='0' then
         if credit_i_in='1' then
            if majority='0' then
               data_ant(7 downto 0) <= data_in;
               data_ant(8)<= '0';
            else
               data_ant(7 downto 0) <= not data_in;
               data_ant(8)<= '1';
            end if;
         end if;
      end if;
   end process;
   dataIn_xor_dataAnt(7 downto 0) <= data_in xor data_ant(7 downto 0);
   dataIn_xor_dataAnt(8) <= data_ant(8);
   tx <= rx;
   clock_tx_out <= clock_tx_in;
   credit_i_out <= credit_i_in;

end BusInv_encoder2;
```

Tabela A.8: Código VHDL do módulo *Majority_9*, componente interno do codificador *Bus-Invert*.

```
Library IEEE,work;
use IEEE.std_logic_1164.all;

entity majority_9 is
   port(
      vet : in std_logic_vector(8 downto 0);
      s4A : out std_logic
   );
end majority_9;


architecture s1 of majority_9 is

signal     s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13,s14,s15,s16     :
std_logic;
signal s1A,s1B,s2A,s2B,s3A,s3B,s3C,s3D,s3E : std_logic;

begin
   s1 <= vet(3) or vet(4);
   s2 <= vet(2) or vet(5);
   s3 <= vet(1) or vet(6);
   s4 <= vet(0) or vet(7);
   s5 <= (vet(8) and s1) or s16;
   s6 <= s1 or vet(8);
   s7 <= (vet(8) and s16) or (s6 and s13);
   s8 <= (vet(5) and vet(2) and s4) or (s2 and s15);
   s9 <= (s3 and s5) or s7;
   s10 <= s4 or s2;
   s11 <= s6 or s3;
   s12 <= (s3 and s10) or s13;
   s13 <= vet(6)and vet(1);
   s14 <= (s11 and s15) or s9;
   s15 <= vet(7) and vet(0);
   s16 <= vet(4) and vet(3);

   s1A <= s2 and s4;
   s1B <= s3 and s6;

   s2A <= s15 or s1A;
   s2B <= s13 or s5 or s1B;

   s3A <= not (vet(8) and s12 and s16);
   s3B <= not (vet(5) and vet(2) and s14);
   s3C <= not (s5 and s10 and s13);
   s3D <= not (s9 and s2A);
   s3E <= not (s8 and s2B);

   s4A <= not (s3A and s3B and s3C and s3D and s3E);

end s1;
```

Tabela A.9: Código VHDL do decodificador *Bus-Invert* com 1 *cluster* e largura de *flit* igual a 8 bits.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity BusInv_decoder is
port(
   clock_rx_in : in std_logic;
   reset : in std_logic;
   data in : in std logic vector(8 downto 0),   -- 9 bits, 8 data + 1
```

```
inv
   rx : in std_logic;
   clock_rx_out : out std_logic;
   data_out : out std_logic_vector(7 downto 0);
   tx : out std_logic;
   credit_o_in : in std_logic;
   credit_o_out : out std_logic
);
end BusInv_decoder;

architecture BusInv_decoder of BusInv_decoder is
begin
   process(reset, clock_rx_in)
   begin
      if reset='1' then
      elsif clock_rx_in'event and clock_rx_in='0' then
         if credit_o_in='1' then
            if data_in(8) = '0' then
               data_out <= data_in(7 downto 0);
            else
               data_out <= not data_in(7 downto 0);
            end if;
         end if;
      end if;
   end process;

   tx <= rx;
   clock_rx_out <= clock_rx_in;
   credit_o_out <= credit_o_in;

end BusInv_decoder;
```

Tabela A.10: Código VHDL do codificador *T-Bus-Invert* com largura de flit igual a 8 *bits*.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity encoder_T_bus_inv_majority_8bits_V2 is
   port(
      clock_tx_in : in std_logic;
      reset : in std_logic;
      data_in : in std_logic_vector(7 downto 0);
      rx : in std_logic;
      clock_tx_out : out std_logic;
      data_out : out std_logic_vector(7 downto 0);
      tx : out std_logic;
      credit_i_in : in std_logic;
      credit_i_out : out std_logic
   );
end encoder_T_bus_inv_majority_8bits_V2;

architecture          encoder T bus inv majority 8bits V2          of
encoder_T_bus_inv_majority_8bits_V2 is

--tamanho do pacote; segundo flit do cabecalho
signal tamanho_pacote : std_logic_vector(7 downto 0);
signal    cont flit payload,    data ant,    data to out,    data cod:
std_logic_vector(7 downto 0);
signal data_to_data_to_out : std_logic_vector(6 downto 0);
type fsm is (S0,S1,S2);
signal EA : fsm;
signal data_to_buff, buff : std_logic_vector(6 downto 0);
signal dataIn_xor_dataAnt : std_logic_vector(7 downto 0);
signal result : std_logic;
signal cont8 : std_logic_vector(2 downto 0);
```

```
begin
   s: entity majority_8
      port map(
         vet => dataIn_xor_dataAnt,
         y => result
      );

   process(reset, clock_tx_in)
   begin
      if reset='1' then
         data_ant <= (others=>'0');
         EA <= S0;
         cont_flit_payload <= (others=>'0');
         cont8 <= (others=>'0');
         tx<='0';
      elsif clock_tx_in'event and clock_tx_in='0' then
         tx <= rx;
         if credit_i_in='1' and rx='1' then
            data_ant <= data_to_out;
            data_out <= data_to_out;
            case EA is
               -- estado inicial, antes de transmissao e transmissao
do primeiro flit
               when S0 =>
                  EA <= S1;
               -- transmissao do segundo flit, tamanho do pacote
               when S1 =>
                  tamanho_pacote <= data_in;
                  cont_flit_payload <= cont_flit_payload + 1;
                  EA <= S2;
               when S2 =>
                  buff <= data_to_buff;
                  cont8 <= cont8 + 1;
                  if cont8 < 7 then
                     cont_flit_payload <= cont_flit_payload + 1;
                  end if;
                  if cont_flit_payload = tamanho_pacote then
                     cont_flit_payload <= (others=>'0');
                     EA <= S0;
                  else
                     EA <= S2;
                  end if;
            end case;
         end if;
      end if;
   end process;

   data_to_buff(0)<= data_in(7) when cont8 = 0
      else  data_in(6) when cont8 = 1
      else  data_in(5) when cont8 = 2
      else  data_in(4) when cont8 = 3
      else  data_in(3) when cont8 = 4
      else  data_in(2) when cont8 = 5
      else  data_in(1) when cont8 = 6;

   data_to_buff(1)<= data_in(7) when cont8 = 1
      else  data_in(6) when cont8 = 2
      else  data_in(5) when cont8 = 3
      else  data_in(4) when cont8 = 4
      else  data_in(3) when cont8 = 5
      else  data_in(2) when cont8 = 6;

   data_to_buff(2)<= data_in(7) when cont8 = 2
      else  data_in(6) when cont8 = 3
      else  data_in(5) when cont8 = 4
      else  data_in(4) when cont8 = 5
      else  data_in(3) when cont8 = 6;

   data_to_buff(3)<= data_in(7) when cont8 = 3
```

```
      else  data_in(6) when cont8 = 4
      else  data_in(5) when cont8 = 5
      else  data_in(4) when cont8 = 6;

   data_to_buff(4)<= data_in(7) when cont8 = 4
      else  data_in(6) when cont8 = 5
      else  data_in(5) when cont8 = 6;

   data_to_buff(5)<= data_in(7) when cont8 = 5
      else  data_in(6) when cont8 = 6;

   data_to_buff(6)<= data_in(7) when cont8 = 6;

   data_to_data_to_out<= data_in(6 downto 0) when cont8 = 0
      else buff(0) & data_in(5 downto 0) when cont8 = 1
      else buff(1 downto 0) & data_in(4 downto 0) when cont8 = 2
      else buff(2 downto 0) & data_in(3 downto 0) when cont8 = 3
      else buff(3 downto 0) & data_in(2 downto 0) when cont8 = 4
      else buff(4 downto 0) & data_in(1 downto 0) when cont8 = 5
      else buff(5 downto 0) & data_in(0) when cont8 = 6
      else buff(6 downto 0);  -- cont8 = 7, envia os 7 bits atrasados

   dataIn xor dataAnt(6  downto  0)  <=  data to data to out(6  downto
0) xor data_ant(6 downto 0);
   dataIn_xor_dataAnt(7) <= data_ant(7);

   data_cod <= '0' & data_to_data_to_out when result = '0'
      else '1' & not data_to_data_to_out;

   data_to_out <= data_in when EA = S0 or EA = S1
      else data_cod;

   clock_tx_out <= clock_tx_in;

   -- pára o recebimento de novos dados enquanto envia os 7 bits
atrasados
   credit_i_out <= '0' when cont8 = 7
      else credit_i_in;

end encoder T bus inv majority 8bits V2;
```

Tabela A.11: Código VHDL do módulo *Majority_8*, componente interno do codificador *T-Bus-Invert*.

```
library IEEE,work;
use IEEE.std_logic_1164.all;

entity majority_8 is
   port(
      vet : in std_logic_vector(7 downto 0);
      y : out std_logic
   );
end majority_8;

architecture a1 of majority_8 is

signal s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12 : std_logic;
begin
   s1 <= vet(2) or vet(3);
   s2 <= (vet(3) and vet(2)) or (vet(4) and s1);
```

```
   s3 <= vet(1) or vet(5);
   s4 <= vet(0) or vet(6);
   s5 <= (vet(5) and vet(1)) or (vet(7) and s3);
   s6 <= s1 or vet(4);
   s7 <= s3 or vet(7);
   s8 <= s12 or s2;
   s9 <= s12 or s5;
   s10 <= (s4 and s6) or s8;
   s11 <= (s4 and s7) or s9;
   s12 <= vet(6) and vet(0);

   y <= (vet(4) and vet(3) and vet(2) and s11) or (vet(7) and vet(5)
and vet(1) and s10) or (s2 and s7 and s12) or (s5 and s6 and s12) or
(s2 and s4 and s5);

end a1;
```

Tabela A.12: Código VHDL do decodificador *T-Bus-Invert* com largura de *flit* igual a 8 bits.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity decoder_temp_bus_inv_8bits_V2 is
   port(
      clock_rx_in : in std_logic;
      reset : in std_logic;
      data_in : in std_logic_vector(7 downto 0);
      rx : in std_logic;
      clock_rx_out : out std_logic;
      data_out : out std_logic_vector(7 downto 0);
      tx : out std_logic;
      credit_o_in : in std_logic;
      credit_o_out : out std_logic
   );
end decoder_temp_bus_inv_8bits_V2;

architecture              decoder temp bus inv 8bits V2              of
decoder_temp_bus_inv_8bits_V2 is

--tamanho do pacote; segundo flit do cabecalho
signal tamanho_pacote : std_logic_vector(7 downto 0);
signal cont_flit_payload, data to out, data decod: std logic vector(7
downto 0);
type fsm is (S0,S1,S2);
signal EA : fsm;
signal data_to_buff, buff : std_logic_vector(6 downto 0);
signal cont8 : std_logic_vector(2 downto 0);

begin

   process(reset, clock_rx_in)
   begin
      if reset='1' then
         EA <= S0;
         cont_flit_payload <= (others=>'0');
         cont8 <= (others=>'0');
         tx<='0';
      elsif clock_rx_in'event and clock_rx_in='1' then
         -- para o envio de novos dados enquanto nao tem um byte
inteiro pra enviar
         if cont8 = 0 and EA = S2 then
            tx <= '0';
         else
            tx <= rx;
         end if;
         if credit_o_in='1' and rx='1' then
```

```
            data_out <= data_to_out;
            case EA is
               -- estado inicial, antes de transmissao e transmissao
do primeiro flit
               when S0 =>
                  EA <= S1;
               -- transmissao do segundo flit, tamanho do pacote
               when S1 =>
                  tamanho_pacote <= data_in;
                  cont_flit_payload <= cont_flit_payload + 1;
                  EA <= S2;
               when S2 =>
                  if data_in(7) = '0' then
                     buff <= data_to_buff;
                  else
                     buff <= not data_to_buff;
                  end if;
                  cont8 <= cont8 + 1;
                  if cont8 > 0 then
                     cont_flit_payload <= cont_flit_payload + 1;
                  end if;
                  if cont_flit_payload = tamanho_pacote then
                     cont_flit_payload <= (others=>'0');
                     EA <= S0;
                  else
                     EA <= S2;
                  end if;
            end case;
         end if;
      end if;
   end process;

   data_to_buff(0)<= data in(0) when cont8 = 0 or cont8 = 1 or cont8 =
2 or cont8 = 3 or cont8 = 4 or cont8 = 5 or cont8 = 6;
   data to buff(1)<= data in(1) when cont8 = 0 or cont8 = 1 or cont8 =
2 or cont8 = 3 or cont8 = 4 or cont8 = 5;
   data_to_buff(2)<= data_in(2) when cont8 = 0 or cont8 = 1 or cont8 =
2 or cont8 = 3 or cont8 = 4;
   data to buff(3)<= data in(3) when cont8 = 0 or cont8 = 1 or cont8 =
2 or cont8 = 3;
   data to buff(4)<= data in(4) when cont8 = 0 or cont8 = 1 or cont8 =
2;
   data_to_buff(5)<= data_in(5) when cont8 = 0 or cont8 = 1;
data_to_buff(6)<= data_in(6) when cont8 = 0;

   data_decod <= data_in(6) & buff when cont8 = 1 and data_in(7) = '0'
      else not data_in(6) & buff when cont8 = 1 and data_in(7) = '1'
      else data_in(6 downto 5) & buff(5 downto 0) when cont8 = 2 and
data_in(7)='0'
      else not data in(6 downto 5)& buff(5 downto 0) when cont8=2 and
data_in(7)='1'
      else data_in(6 downto 4) & buff(4 downto 0) when cont8 = 3 and
data_in(7)='0'
      else not data_in(6 downto 4)& buff(4 downto 0) when cont8=3 and
data_in(7)='1'
      else data in(6 downto 3)& buff(3 downto 0) when cont8 = 4 and
data_in(7) = '0'
      else not data in(6 downto 3)& buff(3 downto 0) when cont8=4 and
data_in(7)='1'
      else data_in(6 downto 2) & buff(2 downto 0) when cont8 = 5 and
data_in(7)='0'
      else not data in(6 downto 2)& buff(2 downto 0) when cont8=5 and
data_in(7)='1'else data in(6 downto 1) & buff(1 downto 0) when cont8 =
6 and data_in(7)='0'
      else not data_in(6 downto 1)& buff(1 downto 0) when cont8=6 and
data_in(7)='1'
      else data in(6 downto 0) & buff(0) when cont8 = 7 and data in(7)
= '0'
      else not data in(6 downto 0) & buff(0) when cont8 = 7 and
```

```
data_in(7) = '1';

   data_to_out <= data_in when EA = S0 or EA = S1
      else data_decod;

   clock_rx_out <= clock_rx_in;
   credit_o_out <= credit_o_in;

end decoder_temp_bus_inv_8bits_V2;
```

Tabela A.13: Código VHDL do codificador *Bus-Invert* com 1 *cluster* e largura de *flit* igual a 16 bits.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity busInv_encoder_16_1cl is
   port(
      clock_tx_in : in std_logic;
      reset : in std_logic;
      data_in : in std_logic_vector(15 downto 0);    -- entrada de 16
bits
      rx : in std_logic;
      clock_tx_out : out std_logic;
      data_out : out std_logic_vector(16 downto 0); -- saida 17 bits,
16 data + 1 inv
      tx : out std_logic;
      credit_i_in : in std_logic;
      credit_i_out : out std_logic
   );
end busInv_encoder_16_1cl;

architecture busInv_encoder_16_1cl of busInv_encoder_16_1cl is

signal data_ant : std_logic_vector(16 downto 0);
signal dataIn_xor_dataAnt : std_logic_vector(16 downto 0);
signal result: std_logic_vector(4 downto 0);
signal vet_1: std_logic_vector(16 downto 0);

begin

   vet_1(15 downto 0) <= dataIn_xor_dataAnt(15 downto 0);
   vet_1(16) <= dataIn_xor_dataAnt(16);

   s1: entity work.soma_17
      port map(
         entrada => vet_1,
         saida => result
      );

   process(reset, clock_tx_in)
   begin
      if reset='1' then
         data_out(16 downto 0) <= (others=>'0');
      elsif clock_tx_in'event and clock_tx_in='1' then
         if credit_i_in='1' then
            if result < 9 then
               data_out(15 downto 0) <= data_in(15 downto 0);
               data_out(16) <= '0';
            else
               data_out(15 downto 0) <= not data_in(15 downto 0);
               data_out(16) <= '1';
            end if;
         end if;
      end if;
   end process;
```

```
   process(reset, clock_tx_in)
   begin
      if reset='1' then
         data_ant <= (others=>'0');
      elsif clock_tx_in'event and clock_tx_in='0' then
         if credit_i_in='1' then
            if result < 9 then
               data_ant(15 downto 0) <= data_in(15 downto 0);
               data_ant(16) <= '0';
            else
               data_ant(15 downto 0) <= not data_in(15 downto 0);
               data_ant(16) <= '1';
            end if;
         end if;
      end if;
   end process;

   dataIn xor dataAnt(15 downto 0) <= data in xor data ant(15 downto
0);
   dataIn_xor_dataAnt(16) <= data_ant(16);

   tx <= rx;
   clock_tx_out <= clock_tx_in;
   credit_i_out <= credit_i_in;

end busInv_encoder_16_1cl;
```

Tabela A.14: Código VHDL do módulo *somador_17*, componente do codificador *Bus-Invert*.

```
library IEEE,work;
use IEEE.std_logic_1164.all;

entity soma_17 is
   port(
      entrada : in std_logic_vector(16 downto 0);
      saida : out std_logic_vector(4 downto 0));
end soma_17;

architecture soma_17 of soma_17 is

signal resA, resB : std_logic_vector(3 downto 0);
signal resULA : std_logic_vector(4 downto 0);

begin
   s1: entity work.somador_8 (s1)
      port map(
         vet => entrada(15 downto 8),
         result => resA);
   s2: entity work.somador_8 (s1)
      port map(
         vet => entrada(7 downto 0),
         result => resB);
   s3: entity work.RippleCarry (RippleCarry)
      port map(
         A => resA,
         B => resB,
         Cin => entrada(16),
         S => resULA);

   saida <= resULA;

end soma_17;
```

Tabela A.15: Código VHDL do decodificador *Bus-Invert* com 1 *cluster* e largura de *flit* igual a 16 bits.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity BusInv_decoder_16_1cl is
   port(
      clock_rx_in : in std_logic;
      reset : in std_logic;
      data_in : in std_logic_vector(16 downto 0); -- 17 bits, 16 data
+ 1 inv
      rx : in std_logic;
      clock_rx_out : out std_logic;
      data_out : out std_logic_vector(15 downto 0);
      tx : out std_logic;
      credit_o_in : in std_logic;
      credit_o_out : out std_logic
   );
end BusInv_decoder_16_1cl;

architecture BusInv_decoder_16_1cl of BusInv_decoder_16_1cl is

begin

   process(reset, clock_rx_in)
   begin
      if reset='1' then
      elsif clock_rx_in'event and clock_rx_in='0' then
         if credit_o_in='1' then
            if data_in(16) = '0' then
               data_out(15 downto 0) <= data_in(15 downto 0);
            else
               data_out(15 downto 0) <= not data_in(15 downto 0);
            end if;
         end if;
      end if;
   end process;

   tx <= rx;
   clock_rx_out <= clock_rx_in;
   credit_o_out <= credit_o_in;

end BusInv_decoder_16_1cl;
```

Tabela A.16: Código VHDL do codificador *Bus-Invert* com 2 *clusters* e largura de *flit* igual a 16 bits.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity busInv_encoder_16_2cl is
   port(
      clock_tx_in : in std_logic;
      reset : in std_logic;
      data_in : in std_logic_vector(15 downto 0);   -- entrada de 16
bits
      rx : in std_logic;
      clock_tx_out : out std_logic;
      data_out : out std_logic_vector(17 downto 0); -- saida 18 bits,
16 data + 2 inv
      tx : out std_logic;
      credit_i_in : in std_logic;
      credit_i_out : out std_logic
   );
end busInv_encoder_16_2cl;

architecture busInv_encoder_16_2cl of busInv_encoder_16_2cl is
```

```
signal data_ant : std_logic_vector(17 downto 0);
signal dataIn_xor_dataAnt : std_logic_vector(17 downto 0);
signal result_1, result_2 : std_logic_vector(3 downto 0);
signal vet_1, vet_2 : std_logic_vector(8 downto 0);

begin

   vet_1(7 downto 0) <= dataIn_xor_dataAnt(7 downto 0);
   vet_1(8) <=dataIn_xor_dataAnt(16);
   vet_2(7 downto 0) <= dataIn_xor_dataAnt(15 downto 8);
   vet_2(8) <=dataIn_xor_dataAnt(17);

   s1: entity work.somador_9
      port map(
         vet => vet_1,
         result => result_1
      );

   s2: entity work.somador_9
      port map(
         vet => vet_2,
         result => result_2
      );

   process(reset, clock_tx_in)
   begin
      if reset='1' then
         data_out(17 downto 0) <= (others=>'0');
      elsif clock_tx_in'event and clock_tx_in='1' then
         if credit_i_in='1' then
            if result_1 < 5 then
               data_out(7 downto 0) <=data_in(7 downto 0);
               data_out(16) <= '0';
            else
               data_out(7 downto 0) <=not data_in(7 downto 0);
               data_out(16) <= '1';
            end if;

            if result_2 < 5 then
               data_out(15 downto 8) <= data_in(15 downto 8);
               data_out(17) <= '0';
            else
               data_out(15 downto 8) <= not data_in(15 downto 8);
               data_out(17) <= '1';
            end if;
         end if;
      end if;
   end process;

   process(reset, clock_tx_in)
   begin
      if reset='1' then
         data_ant <= (others=>'0');
      elsif clock_tx_in'event and clock_tx_in='0' then
         if credit_i_in='1' then
            if result_1 < 5 then
               data_ant(7 downto 0) <= data_in(7 downto 0);
               data_ant(16) <= '0';
            else
               data_ant(7 downto 0) <= not data_in(7 downto 0);
               data_ant(16) <= '1';
            end if;
            if result_2 < 5 then
               data_ant(15 downto 8) <= data_in(15 downto 8);
               data_ant(17) <= '0';
            else
               data_ant(15 downto 8) <= not data_in(15 downto 8);
               data_ant(17) <= '1';
            end if;
```

```
         end if;
      end if;
   end process;

   dataIn xor dataAnt(15 downto 0) <= data in xor data ant(15 downto
0);
   dataIn_xor_dataAnt(17 downto 16) <= data_ant(17 downto 16);

   tx <= rx;
   clock_tx_out <= clock_tx_in;
   credit_i_out <= credit_i_in;

end busInv encoder 16 2cl;
```

Tabela A.17: Código VHDL do decodificador *Bus-Invert* com 2 *clusters* e largura de *flit* igual a 16 bits.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity BusInv_decoder_16_2cl is
   port(
      clock_rx_in : in std_logic;
      reset : in std_logic;
      data_in : in std_logic_vector(17 downto 0); -- 18 bits, 16 data
+ 2 inv
      rx : in std_logic;
      clock_rx_out : out std_logic;
      data_out : out std_logic_vector(15 downto 0);
      tx : out std_logic;
      credit_o_in : in std_logic;
      credit_o_out : out std_logic
   );
end BusInv_decoder_16_2cl;

architecture BusInv_decoder_16_2cl of BusInv_decoder_16_2cl is

begin

   process(reset, clock_rx_in)
   begin
      if reset='1' then
      elsif clock_rx_in'event and clock_rx_in='0' then
         if credit_o_in='1' then
            if data_in(16) = '0' then
               data_out(7 downto 0) <= data_in(7 downto 0);
            else
               data_out(7 downto 0) <= not data_in(7 downto 0);
            end if;
            if data_in(17) = '0' then
               data_out(15 downto 8) <= data_in(15 downto 8);
            else
               data_out(15 downto 8) <= not data_in(15 downto 8);
            end if;
         end if;
      end if;
   end process;

   tx <= rx;
   clock_rx_out <= clock_rx_in;
   credit_o_out <= credit_o_in;

end BusInv_decoder_16_2cl;
```

Tabela A.18: Código VHDL do codificador *T-Bus-Invert* com largura de *flit* igual a 16 bits.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity Encoder_T_Bus_Inv_16 is
   port(
      clock_tx_in : in std_logic;
      reset : in std_logic;
      data_in : in std_logic_vector(15 downto 0);
      rx : in std_logic;
      clock_tx_out : out std_logic;
      data_out : out std_logic_vector(15 downto 0);
      tx : out std_logic;
      credit_i_in : in std_logic;
      credit_i_out : out std_logic
   );
end Encoder_T_Bus_Inv_16;

architecture Encoder_T_Bus_Inv_16 of Encoder_T_Bus_Inv_16 is

signal tx2, credit_i_out2: std_logic;

begin

   d1: entity work.encoder_T_bus_inv_majority_8bits_V2
      port map(
         clock_tx_in => clock_tx_in,
         reset => reset,
         data_in => data_in(15 downto 8),
         rx => rx,
         clock_tx_out =>clock_tx_out,
         data_out => data_out(15 downto 8),
         tx => tx,
         credit_i_in => credit_i_in,
         credit_i_out =>credit_i_out
      );

   d2: entity work.encoder_T_bus_inv_majority_8bits_V2
      port map(
         clock_tx_in => clock_tx_in,
         reset => reset,
         data_in => data_in(7 downto 0),
         rx => rx,
         clock_tx_out =>clock_tx_out,
         data_out => data_out(7 downto 0),
         tx => tx2,
         credit_i_in => credit_i_in,
         credit_i_out =>credit_i_out2
      );

end Encoder_T_Bus_Inv_16;
```

Tabela A.19: Código VHDL do decodificador *T-Bus-Invert* com largura de *flit* igual a 16 bits.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity Decoder_T_Bus_Inv_16 is
   port(
      clock_rx_in : in std_logic;
      reset : in std_logic;
      data_in : in std_logic_vector(15 downto 0);
      rx : in std_logic;
```

```
      clock_rx_out : out std_logic;
      data_out : out std_logic_vector(15 downto 0);
      tx : out std_logic;
      credit_o_in : in std_logic;
      credit_o_out : out std_logic
   );
end Decoder_T_Bus_Inv_16;

architecture Decoder_T_Bus_Inv_16 of Decoder_T_Bus_Inv_16 is
signal tx2, credit_o_out2: std_logic;

begin

   d1: entity work.decoder_temp_bus_inv_8bits_V2
      port map(
         clock_rx_in => clock_rx_in,
         reset => reset,
         data_in => data_in(15 downto 8),
         rx => rx,
         clock_rx_out =>clock_rx_out,
         data_out => data_out(15 downto 8),
         tx => tx,
         credit_o_in => credit_o_in,
         credit_o_out =>credit_o_out
      );

   d2: entity work.decoder_temp_bus_inv_8bits_V2
      port map(
         clock_rx_in => clock_rx_in,
         reset => reset,
         data_in => data_in(7 downto 0),
         rx => rx,
         clock_rx_out =>clock_rx_out,
         data_out => data_out(7 downto 0),
         tx => tx2,
         credit_o_in => credit_o_in,
         credit_o_out =>credit_o_out2
      );

end Decoder_T_Bus_Inv_16;
```