ANDRÉ BARBOZA MACIEL GOMES

ÔMEGA ASSIMÉTRICA: UMA NOVA REDE DE INTERCONEXÃO PARA DEPURAÇÃO PÓS-SILÍCIO

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

VIÇOSA
MINAS GERAIS - BRASIL
2015

Ficha catalográfica preparada pela Biblioteca Central da Universidade Federal de Viçosa - Campus Viçosa

T
G633o
2015

Gomes, André Barboza Maciel, 1986-
Ômega Assimétrica: Uma nova rede de interconexão para depuração pós-silício / André Barboza Maciel Gomes. – Viçosa, MG, 2015.
xiv, 107f. : il. (algumas color.) ; 29 cm.

Inclui apêndices.
Orientador: José Augusto Miranda Nacif
Dissertação (mestrado) – Universidade Federal de Viçosa.
Referências bibliográficas: f.59–64.

1. Circuitos integrados digitais. 2. Eletrônica digital. 3. Redes Ômega. 4. Redes de Interconexão. 5. *Trace Buffer*. I. Universidade Federal de Viçosa. Departamento de Informática. Programa de Pós-Graduação em Ciência da Computação. II. Título.

CDD 22.ed 621.381

ANDRÉ BARBOZA MACIEL GOMES

ÔMEGA ASSIMÉTRICA: UMA NOVA REDE DE INTERCONEXÃO PARA DEPURAÇÃO PÓS-SILÍCIO

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

APROVADA: 24 de Abril de 2015.

Ricardo dos Santos Ferreira
(Coorientador)

Omar Paranaíba Vilela Neto

Frank Sill Torres

José Augusto Miranda Nacif
Orientador

Esta dissertação é dedicada aos meus pais,
Francisco e Léa.

Agradecimentos

Agradeço a Deus pela boas condições de vida, saúde e pela benção de ter uma família maravilhosa. Em especial agradeço aos meus pais Francisco e Léa que me ajudaram muito para conseguir tudo que tenho hoje. Eles me apoiaram em todos os momentos da minha vida. Também agradeço a todos meus familiares que puderam compreender os longos períodos em que estive afastado para me dedicar aos estudos. Agradeço à minha namorada Angelica Lima por também compreender todo esforço e dedicação que é necessário para chegar até aqui. Foram diversos finais de semana que precisei trabalhar para concluir esse trabalho e para submissão de artigos.

Agradeço aos meus orientadores Prof. José Nacif e Prof. Ricardo Ferreira, que foram fundamentais para o direcionamento da pesquisa. Eles foram capazes de me guiar com seus conhecimentos, impulsionaram e acreditaram no meu trabalho. Agradeço também aos professores e funcionários do Departamento de Informática (DPI) da Universidade Federal de Viçosa (UFV). Adquiri muitos conhecimentos nas aulas dos professores André dos Santos, José Arroyo, Alcione Oliveira, José Braga, José Nacif e Ricardo Ferreira. Agradeço ao funcionário Altino de Sousa e ao coordenador do DPI Prof. Jugurta Lisboa pelo suporte enquanto estava no Campus Florestal. Agradeço também à todos os professores que contribuíram com minha formação intelectual.

Agradeço àqueles que contribuíram com o desenvolvimento do trabalho. Agradeço ao Fredy Alves aluno de iniciação científica do curso de Ciência da Computação no Campus Florestal por sua contribuição na realização de experimentos. Agradeço ao Frank Torres do Departamento de Engenharia Eletrônica (DELT) da Universidade Federal de Minas Gerais (UFMG), e a instituição UFMG por ceder espaço físico e recursos computacionais para a utilização do software necessário nos experimentos realizados.

Agradeço à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), o Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), à Universidade Federal de Viçosa (UFV) e à Fundação de Amparo à Pesquisa do Estado de Minas Gerais (FAPEMIG) pelo apoio financeiro.

"If you can dream it, you can do it.
(Walt Disney)

Sumário

Lista de ilustrações

Lista de algoritmos

Lista de códigos

Lista de tabelas

Lista de abreviaturas e siglas

APE	Automação de Projeto de Eletrônico
CAD	*Computer Aided Design*
CC	Carregador de Configuração
CeD	Circuito em Depuração
CID	Circuito Integrado Digital
DeV	Dispositivo em Verificação
ETA	Equipamento de Teste Automatizado
FPGA	*Field-programmable gate array*
LDH	Linguagem de Descrição de Hardware
MEF	Máquina de Estados Finitos
Mux Tree	*Pipelined Multiplexer Tree*
PI	Propriedade Intelectual
PpD	Projeto para Depuração
RIM	Rede de Interconexão Multiestágio
RTL	*Register Transfer Level*
SAT	Problema de Satisfazibilidade Booleana
SoC	*System On Chip*

Resumo

Gomes, André Barboza Maciel, M.Sc., Universidade Federal de Viçosa, Abril de 2015. **Ômega Assimétrica: Uma nova rede de interconexão para depuração pós-silício**. Orientador: José Augusto Miranda Nacif. Coorientador: Ricardo dos Santos Ferreira

As atuais técnicas de verificação pré-silício não garantem que na primeira fabricação, um circuito complexo seja produzido sem erros. Alguns erros só acontecem quando o circuito é executado em sua velocidade real, assim, projetistas utilizam técnicas de depuração pós-silício para monitorar o circuito e capturar erros que ocorrem somente depois de milhões de ciclos de *clock*. Esse processo se tornou essencial e consome em média 35% do tempo de ciclo de desenvolvimento de um Circuito Integrado Digital. Na depuração pós-silício a limitação de observabilidade é um problema desafiador, e para identificar a causa de um erro o projetista inclui uma infraestrutura para depuração. Na técnica *Trace Buffer* alguns valores de sinais são armazenadas em uma memória de rastreamento, extraídos e analisados. O tamanho da memória de rastreamento restringe o número de sinais que podem ser analisados. A escolha do conjunto de sinais é essencial, porém é realizada antes mesmo de qualquer identificação de erro no projeto. Para possibilitar o monitoramento de diversos conjuntos de sinais, na indústria é utilizado uma rede de interconexão composto por multiplexadores encadeados (Mux Tree), que permite o projetista monitorar um subconjunto de todos os sinais que podem ser explorados. A arquitetura dessa rede não permite a seleção de qualquer conjunto de sinais, uma vez que sinais que passam pelos mesmos multiplexadores não podem ser monitorados juntos. Nesse trabalho é proposto uma nova rede de interconexão baseada na tradicional rede Ômega. A rede proposta pode ser utilizada como um dispositivo de interconexão para conectar os sinais monitorados à memória de rastreamento. Nesse trabalho é demonstrado que a rede Ômega assimétrica proposta pode reduzir em 4,5 vezes a taxa de bloqueio, ao custo de aumentar em 21% a área, se comparado à rede de Mux Tree. A rede Ômega assimétrica pode ser gerada utilizando a ferramenta proposta nesse trabalho, Vericonn, que também é capaz de gerar em Verilog outras redes assimétricas como: Redes Mux Tree, Clos e Crossbars.

Abstract

Gomes, André Barboza Maciel, M.Sc., Universidade Federal de Viçosa, April, 2015. **Asymmetrical Omega: A new interconnection network for post-silicon debug**. Adviser: José Augusto Miranda Nacif. Co-adviser: Ricardo dos Santos Ferreira.

Current pre-silicon verification techniques can not guarantee error free designs for complex integrated circuits during their first fabrication. Some errors are only uncovered when the device is running at full clock speed, thus, designers use post-silicon debug techniques to monitor the device, capturing errors that occur only after millions of clock cycles. This process has become essential and on average consumes 35% of the Digital Integrated Circuit development cycle. Observability limitation is a challenging problem in post-silicon debug, so to identify the root cause of an error, designers include an infrastructure for debug. In Trace Buffer technique, some signal values are stored in a Trace Buffer memory, dumped, and then analyzed. The Trace Buffer memory size limits the number of signals that can be analyzed. Choosing the signal set is an essential step, but it must be done prior to the identification of any design errors. To enable the monitoring of many sets of signals, industry uses an interconnection network composed by pipelined multiplexers (Mux Trees) that allows designers to monitor a signal subset from all tapped signals. The architecture of this network does not allow any signal subset because signals passing through the same multiplexers can not be monitored together. In this work, we propose a novel asymmetric network, based on the traditional Omega Network. We propose to use this network as an interconnection fabric to connect the monitored signals to the trace buffer. We demonstrate that our Asymmetric Omega Network is 4.5 times more effective reducing the blocking rate at the cost of 21% area overhead compared to Mux Trees. The proposed network can be generated with our tool, Vericonn, which is also capable to create others asymmetric networks like: Mux Trees, Clos Networks and Crossbars in Verilog HDL.

1 Introdução

Novas tecnologias no processo de desenvolvimento e fabricação permitem a redução no tamanho dos Circuitos Integrados Digitais (CIDs) mesmo que mais funcionalidades sejam incluídas em um único *System On Chip* (SoC). Esse desenvolvimento implica em um aumento na complexidade dos sistemas digitais, o que torna difícil a primeira fabricação sem erros (VERMEULEN; GOEL, 2002). Atrasos nos lançamentos de CIDs podem causar prejuízos de até 93% (VERMEULEN; GOOSSENS, 2014 apud IBM, 2006, p. 5–6). Empresas e investidores pretendem extrair o máximo em lucro e qualidade, e para isso, os produtos devem ser lançados dentro do prazo. Isso torna a verificação um dos estágios mais desafiadores de um ciclo de desenvolvimento de CIDs, pois essa é fase que antecede o lançamento.

Durante o processo de desenvolvimento de um CID, cada estágio passa por um processo de verificação. Na verificação pré-silício, um conjunto de métodos é utilizado para capturar e corrigir erros funcionais do projeto antes da fabricação do CID. Essa verificação é realizada por simulação no circuito, verificação formal e emulação. Utilizando a simulação, o projetista tem uma grande observabilidade do circuito, sendo possível avaliar cada sinal do CID. No entanto, esse processo é muitas vezes mais lento do que o circuito fabricado sendo executado em velocidade real. Dessa forma, a simulação avalia o comportamento do circuito por um número limitado de ciclos de *clock*.

Ao utilizar um método de verificação formal, é possível verificar por completo um circuito utilizando provas matemáticas (KERN; GREENSTREET, 1999). No entanto, para um sistema complexo, com milhões de transistores, essa técnica é impraticável. Para detectar erros em circuitos dessa grandeza, a verificação pré-silício não consegue garantir que não haverá erros após a produção do CID. Para capturar mais erros, um estágio de validação ocorre após a fabricação do circuito. Essa etapa é chamada de verificação pós-silício.

Na verificação pós-silício, o funcionamento do dispositivo manufaturado é avaliado utilizando aplicações e ambientes reais, sendo executado na velocidade que foi projetado. A validação do circuito pós-silício envolve quatro passos principais: *1)* detectar o problema; *2)* reduzir a localização do problema para uma região menor; *3)* identificar a causa do problema; *4)* consertar ou ignorar o problema (MITRA; SESHIA; NICOLICI, 2010). Esse processo se tornou essencial e consome em média 35% do tempo de ciclo de desenvolvimento de um CID (CHANG; MARKOV; BERTACCO, 2007).

Na depuração pós-silício, a limitação de observabilidade é um problema desafiador. A verificação pós-silício impõe restrições na observabilidade de sinais, impedindo o

projetista de capturar e avaliar os valores de sinais no tempo, ao contrário da verificação pré-silício. Para possibilitar a depuração pós-silício, o projetista inclui uma infraestrutura para depuração. Essa infraestrutura pode ser classificada em: *Scan Chains* ou *Trace Buffer* (KO; NICOLICI, 2010).

Ao utilizar o método de *Scan-Chain* é possível ter acesso à todos os valores dos registradores no circuito em depuração. No entanto, o método exige a interrupção da execução do CID. Dessa forma, essa solução exclui a possibilidade de observar o comportamento do circuito durante um período de tempo. Na indústria, a técnica *Trace Buffer* é bastante utilizada na depuração pós-silício (LIU; XU, 2014, p. 5). Essa técnica consiste em incluir uma memória intermediária para monitorar os valores de alguns sinais, possibilitando a análise dos dados por um determinado período. Após a coleta dos dados armazenados na memória, o projetista pode analisar o comportamento do circuito por um período de tempo, aumentando a observabilidade do CID. No entanto, ao utilizar o *Trace Buffer* poucos sinais podem ser monitorados, pois a memória intermediária é restrita pelos custos adicionais no projeto. Como consequência o processo de depuração sofre com a falta de observabilidade, tornando o processo de validação difícil e demorado mesmo para um circuito com poucos ou sem erros (ABRAMOVICI et al., 2006).

Para obter melhores resultados, o projetista deve escolher um bom conjunto de sinais para monitorar. A maior dificuldade para a realização dessa tarefa, é que a seleção do conjunto de sinais deve ocorrer antes da identificação de qualquer erro no projeto. Geralmente o projetista faz suposições *ad hoc* para a probabilidade de localizar erros, utiliza métricas de cobertura de código e asserções para selecionar o conjunto de sinais baseado na complexidade do CID. Em um projeto complexo, um bom conjunto de sinais pode necessitar a interação de vários especialistas (HUNG; WILTON, 2011).

Outra forma para selecionar quais sinais incluir no projeto de depuração é utilizar um algoritmo para fazer a seleção automática de sinais. Trabalhos recentes com o de Basu e Mishra (2013) e Hung e Wilton (2013), propõem algoritmos baseados na probabilidade de restauração de dados provenientes de outros sinais (KO; NICOLICI, 2009). Um algoritmo de seleção automática de sinais pode ajudar projetistas a produzir uma lista que pode ser revisada e melhorada. O algoritmo pode até mesmo produzir uma seleção melhor que um especialista ao reconhecer relacionamentos complexos.

Independente da forma que a seleção de sinais é realizada, algumas combinações de sinais podem ficar indisponíveis quando há um bloqueio na rede de interconexão conectada à memória do *Trace Buffer*. Esses sinais indisponíveis podem ser cruciais para o entendimento e correção de erros. No pior caso o projetista é forçado a incluir os sinais e fabricar novamente[1] o CID para concluir o processo de depuração. Esta situação aumenta o tempo de entrega para o mercado e reduz a receita.

[1] O processo de refabricação é chamado de *respin*.

Ao lidar com projetos com milhões de portas lógicas, pode ser necessário explorar milhares de sinais para obter informações de erro. Devido aos custos da infraestrutura de depuração, apenas alguns sinais podem ser monitorados por vez. A solução padrão da indústria para essa tarefa é a rede *Pipelined Multiplexer Tree* (Mux Tree)(ABRAMOVICI, 2008). Essa rede de interconexão permite que sinais sejam dinamicamente selecionados à cada depuração. Assim, diferentes subconjuntos de sinais podem ser monitorados. A rede de interconexão Mux Tree é composta apenas por multiplexadores 2x1, que são encandeados e formam uma rede com poucos estágios e baixo custo de área.

A arquitetura da rede Mux Tree não permite a seleção de qualquer conjunto de sinais. Isso ocorre pois sinais que passam pelo mesmo encadeamento de multiplexadores não podem ser monitorados juntos. Na rede de interconexão Mux Tree o bloqueio atinge em média 36% em uma seleção arbitrária de sinais [2]. No caso de um ambiente determinístico, seria possível capturar os valores de todos os sinais desejados executando a depuração várias vezes. Assim, pode-se obter a observabilidade total do conjunto de sinais desejado economizando com custos na infraestrutura de depuração. No entanto, em um ambiente real é difícil ter o controle necessário para repetir o experimento diversas vezes. A falta de controle do ambiente real causa resultados diferentes a cada novo teste, isto ocorre por domínios de *clock* assíncronos, certas condições elétricas e algumas condições do ambiente de execução (DEORIO; KHUDIA; BERTACCO, 2011; MITRA; SESHIA; NICOLICI, 2010).

Nesse trabalho é proposta uma nova rede de interconexão multiestágio para depuração pós-silício. Essa rede de interconexão é uma versão assimétrica da tradicional rede Ômega (LAWRIE, 1975). Comparado à Mux Tree, a rede Ômega assimétrica apresenta uma taxa de bloqueio significantemente menor, em contrapartida, um custo de área levemente maior. As **principais contribuições** desse trabalho são:

- Utilização de redes Ômega assimétricas como rede de interconexão para depuração pós-silício. A rede proposta é comparada com outras redes em relação ao custo de área e taxa de bloqueio.
- Descrição e algoritmo para geração de redes Ômega assimétricas de tamanhos arbitrários.
- Ferramenta Vericonn que pode ser utilizada para construir dispositivos de redes de interconexão para depuração pós-silício. A Vericonn é capaz de gerar redes assimétricas de tamanho arbitrários tais como: Crossbar, Mux Trees, redes Clos, e a rede Ômega proposta em Linguagem de Descrição de Hardware (LDH) Verilog.

[2] Resultados obtidos através de experimentos e apresentados na seção 4.

O trabalho está organizado da seguinte forma: No Capítulo 2 é apresentada a contextualização do problema, o processo de desenvolvimento e verificação de CIDs e uma contextualização sobre redes de interconexão. No Capítulo 3 é apresentado o método para a construção da rede Ômega assimétrica e a ferramenta Vericonn para gerar redes assimétricas de tamanhos arbitrários. No Capítulo 4 são apresentados os resultados para validar a rede proposta. Conclusões e trabalhos futuros são apresentados no Capítulo 5.

2 Contextualização

As atuais tecnologias para desenvolvimento e produção de CIDs permitem que mais funcionalidades possam ser adicionadas aos chips. A melhoria contínua desses processos possibilitam que mesmo incluindo novas funcionalidades, haja redução do tamanho do CID. Porém, isso também traz complicações, como o aumento da complexidade do circuito à medida que o número de funcionalidades aumenta. O aumento de complexidade nos circuitos também aumenta o esforço necessário verificar e corrigir de erros. Com a dinâmica do mercado, a demora na entrega do produto é determinante para perda de participação no mercado. Para realizar a comparação, o mercado é segmentado em três níveis de dinamismo (rapidez no lançamento de novos produtos). Em um mercado muito dinâmico, três meses de atraso na entrega do produto significa a perda de cerca de 25% do lucro estimado, 47% em 6 meses e 93% em 12 meses. Para um mercado moderadamente dinâmico, as perdas variam de 17% em 3 meses a 70% em 12 meses. Em um mercado menos dinâmico, as perdas variam de 15% em 3 meses até 50% em 12 meses (VERMEULEN; GOOSSENS, 2014 apud IBM, 2006, p. 5–6).

2.1 Processo de desenvolvimento de um circuito integrado digital

O processo de desenvolvimento de CIDs inclui as fases de especificação, implementação e manufatura. Na Figura 1 é apresentado o fluxograma do processo. A fase de especificação é semelhante ao processo de especificação de software. Uma equipe composta por analistas funcionais cria um documento que contém especificações funcionais do dispositivo. Essa especificação é utilizada para definir o comportamento do dispositivo que é gradualmente refinada. Na Figura 1, os triângulos representam as alternativas para a implementação a partir de um determinado ponto. Logo, quanto maior a abstração, maiores são as alternativas para implementação.

Após a especificação inicial, são utilizadas técnicas para determinar algumas diretrizes sobre o restante do projeto (*back-of-the-envelope calculation*). Conforme mostra a figura, esse direcionamento inicial já começa a guiar os próximos estágios, e consequentemente limita alternativas na implementação. Com esse modelo estimado é criado uma implementação abstrata do circuito. No próximo passo é iniciada implementação do circuito em *Register Transfer Level* (RTL), que então é utilizado para geração do layout do CID. O CID é fabricado na última fase do processo. Repare que os custos de implementação aumentam à medida que a abstração do modelo diminui.

Na fase de implementação são utilizadas linguagens de descrição de hardware ou linguagens de alto nível. As linguagens de alto nível geralmente são subconjuntos de

Figura 1 – Processo de desenvolvimento de um Circuito Integrado. Retirado de (KIENHUIS, 1999).

linguagens C/ANSI, C++, ou SystemC (MARTIN; SMITH, 2009; IEEE..., 2012). A linguagem em alto nível permite que operações sobre dados e a comunicação entre componentes sejam descritos sem uma implementação real de módulos e barramentos. Já nas linguagens de descrição de hardware como Verilog (IEEE..., 2006) e VHDL (IEEE..., 2009), a implementação contém um nível maior de detalhes, o RTL. No RTL, o circuito é descrito pelo fluxo de dados e por operações lógicas entre registradores.

Ferramentas de Automação de Projetos de Eletrônicos (EADs) são utilizadas para transformar a descrição em RTL ou de alto nível para a descrição em nível de portas lógicas (*netlists*). O resultado dessa síntese é então utilizado por outros softwares, que fazem o posicionamento de cada componente do circuito (*placement*) e o roteamento para gerar o *layout* físico. A última fase é a manufatura do dispositivo, que inclui a confecção da fotomáscara e possui vários processos químicos até o circuito final (LIU; XU, 2014).

2.2 Processo de verificação e validação

Na fase de implementação são aplicados procedimentos para detectar e corrigir erros. Essa etapa de verificação é chamada de verificação pré-silício. A técnica mais utilizada para detecção de erros nesta fase é a simulação (TALUPUR, 2011). Na simulação é possível avaliar o valor de cada sinal do circuito. No entanto esse processo é bastante lento, e a escolha dos cenários de testes são criados através de análises de cobertura do circuito. Casos de teste são criados especificamente para analisar determinados comportamentos (YANG; WILLE; DRECHSLER, 2014). Devido ao número exponencial de estímulos que podem ser geradas para a verificação do circuito, pode-se considerar que a simulação de todos os estados é impraticável (ROOTSELAAR; VERMEULEN, 1999; YANG; WILLE; DRECHSLER, 2014).

Com a técnica de verificação formal o circuito é complemente verificado por provas matemáticas (KERN; GREENSTREET, 1999). No entanto, para circuitos mais complexos esse método torna-se também impraticável devido ao tempo necessário para realizar a prova. Se houver erros na especificação das provas matemáticas, a verificação também não será válida. Ao utilizar ambas as técnicas de simulação e verificação formal, o projetista pode eliminar grande parte dos erros.

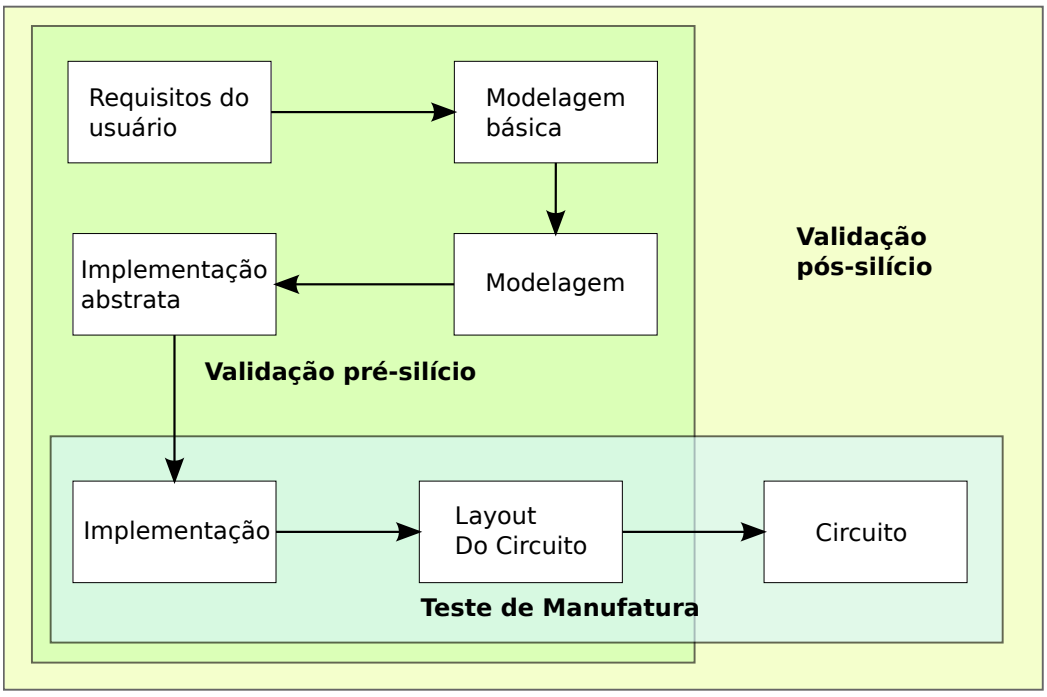


Figura 2 – Fase de implementação de um Circuito Integrado.

Fonte: Figura adaptada de (VERMEULEN; GOOSSENS, 2014, p. 12).

Conforme pode ser observado na Figura 2, a **validação pré-silício** envolve todas as etapas antes da fabricação. Cada etapa de verificação busca por divergências que podem ocorrer nas transformações entre as etapas do desenvolvimento do CID. Esse processo

entre etapas é chamada de **verificação pré-silício**. Assim, como mostra a Figura 1, a cada refinamento o projeto se torna mais caro, e a cada refinamento, o espaço para exploração de alternativas é reduzido.

O desenvolvimento de ferramentas e processos para a fabricação de circuitos integrados digitais permitiram que sistemas complexos possam ser colocados em um único *chip*. São projetos com bilhões de transistores, como é o caso da terceira geração de processadores Intel Core (Intel Corporation, 2012). Para esses circuitos, os processos de verificação como simulação, verificação formal, analise estática de temporização e emulação não conseguem garantir um sistema sem erros após a fabricação (ROOTSELAAR; VERMEULEN, 1999). Esses erros não são detectados na fase pré-silício porque apenas o modelo do circuito é testado, e não o silício em si. Os curtos prazos para lançar produtos no mercado, aliada à grande complexidade dos circuitos, tornam improvável a tarefa de produzir um CID sem erros já na primeira fabricação (VERMEULEN; GOEL, 2002).

O processo para se obter um CID de qualidade é um grande desafio. O aumento na complexidade e a redução no tamanho dos CIDs é aliada a um modelo de desenvolvimento impreciso. Isto resulta em dois tipos de erros que aparecem durante a verificação pós-silício: funcionais e elétricos. Enquanto CIDs com erros funcionais apresentam erros em todas as unidades produzidas, erros elétricos ocorrem somente em algumas unidades. Alguns erros elétricos são decorrentes de interferência entre fios, variações térmicas, e variações na suplementação de energia. Esses erros podem ocorrer devido a alguma variação no processo de fabricação ou em determinadas condições de operação do chip. Erros elétricos não podem ser corrigidos por mudanças no projeto do CID. Conforme mostra a Figura 2, o **teste de manufatura** valida apenas a fabricação do circuito. Um erro de manufatura é detectado através de testes realizados pelo Equipamento de Teste Automatizado (ETA), que injeta alguns sinais no circuito e captura os dados em determinados registradores. Os dados capturados são comparados com o padrão esperado (GAO; LISHERNESS; CHENG, 2011). Já o processo de **validação pós-silício** é uma etapa para localização e correção de erros que estão associados ao comportamento de circuito em relação à especificação funcional.

2.2.1 Paradigmas de depuração de circuitos integrados digitais

Para detectar erros funcionais, vários paradigmas de depuração de circuitos foram desenvolvidos, tais como (ABRAMOVICI et al., 2006): captura de sinal e depuração baseada em asserção. Esses paradigmas também podem ser combinados em um processo de validação.

No paradigma de captura de sinais, uma rede de interconexão é configurada para capturar concorrentemente um conjunto de sinais e armazená-los em uma memória de armazenamento intermediária. Os sinais são conectados a um monitor de depuração, que

tem a tarefa de iniciar e parar a gravação na memória de armazenamento intermediária. O controlador da memória de armazenamento intermediária pode ser configurado para gravar os sinais até que a memória esteja cheia, ou pode gravar os dados em uma fila circular. O conteúdo da memória com valores dos sinais no tempo deve então ser copiado e analisado. Em virtude da limitação do tamanho da memória poucos sinais podem ser analisados por vez (ABRAMOVICI et al., 2006).

O paradigma de asserções é bastante utilizado na verificação pré-silício. Uma asserção utiliza a relação entre sinais para detectar um erro, ao contrário da abordagem de captura de sinais que depende de uma análise manual e intensiva. A utilização de asserções pode encurtar por vários ciclos de *clock* a localização de um determinado erro. Quando uma asserção dispara é bem possível que o erro esteja próximo ao local que a asserção está instanciada. Isto também implica na redução do espaço de busca pelo erro e aumento da observabilidade. Porém, diferentemente da verificação pré-silício que permite a inclusão de um número muito extenso de asserções, na verificação pós-silício a inclusão de asserções é limitada pela quantidade de área extra disponível. No entanto, a verificação pós-silício com asserções traz benefícios de verificação caixa-branca para o circuito fabricado (NACIF et al., 2003; NACIF; PAULA, 2003). Além da utilização na verificação pré-silício e na depuração pós-silício, o paradigma de asserções já foi proposto em um contexto de monitoramento de circuitos (NACIF, 2004) para *Field-programmable gate array*s (FPGAs). Ao monitorar um circuito dessa forma, é possível identificar alguns erros, corrigir e reprogramar o FPGA.

2.3 Infra-estrutura de depuração pós-silício

Para extrair e analisar as informações dos sinais na depuração pós-silício, uma infraestrutura de Projeto para Depuração (PpD) deve ser adicionada ao CID. Idealmente, essa infraestrutura deve prover condições para explorar e rastrear todos os sinais do Circuito em Depuração (CeD) no tempo. No entanto, o custo de área imposto para oferecer tal condição ideal é geralmente proibitivos. Portanto, diferentes técnicas foram propostas para prover uma infraestrutura de PpD com um custo de área aceitável. Essas técnicas podem ser classificadas em: *Scan Chains* e *Trace Buffer*.

Para comparar as duas técnicas, deve-se levar em consideração os escopos temporal e espacial. O escopo temporal determina por quanto tempo é possível observar o conjunto de sinais. O escopo espacial determina a abrangência dos sinais observados. Essas duas métricas estão relacionadas ao volume de dados que podem ser capturados durante o processo de depuração. Cada técnica possui vantagens e desvantagens, além de necessitarem de diferentes mecanismos de hardware para dar suporte à depuração.

2.3.1 Técnica *Scan Chains*

Na técnica de *Scan Chains*, registradores, *Flip-Flops* ou elementos sequenciais são ligados formando um encadeamento. Esse encadeamento permite que os dados sejam extraídos ou inseridos de forma serial, aumentando a observabilidade do circuito (NARAYANAN; DAS, 1997). Na Figura 3 é apresentado o encadeamento dos registradores de depuração, chamados de *Scan Flip-Flops*. Nos *Scan Flip-Flops*, são adicionados sinais nos *Flip-Flops* para controle da entrada e saída e controle do encadeamento. Na Figura 3, além das entradas (**D**) e saídas (**Q**) convencionais, o sinal de escaneamento habilitado (**SE**) controla o dado de entrada, que pode ser a entrada convencional ou o dado proveniente do encadeamento (**SI**).

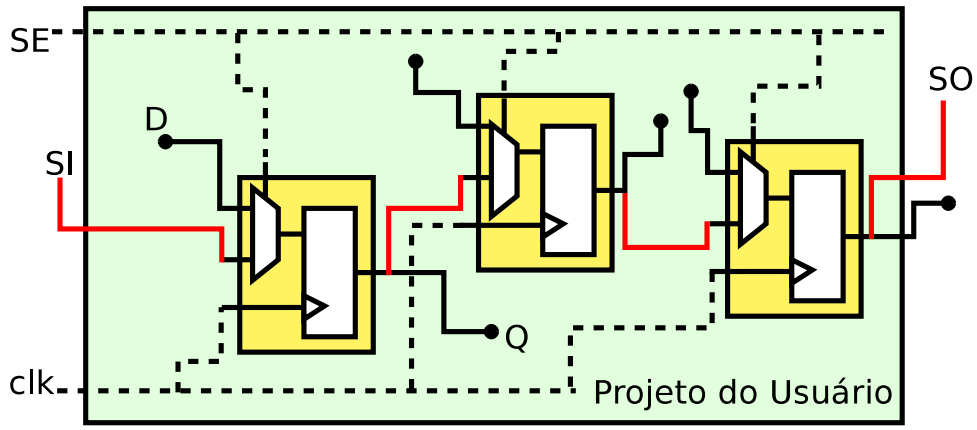


Figura 3 – Método *Scan chain*. Diagrama de blocos de uma infraestrutura de depuração pós-silício com *Scan chain*.

Essa técnica é capaz de capturar cada valor dos sinais dos registradores em um determinado ciclo. O valor é encadeado através da saída (**SO**) e então pode ser recuperado e analisado depois do processo de descarregamento dos dados. Para executar o processo de descarregamento, o projetista deve parar o sistema, tornando esta técnica inadequada para capturar vários ciclos consecutivos (ROSINGER; AL-HASHIMI; NICOLICI, 2004). Assim, a técnica de *Scan Chains* apresenta um escopo espacial completo, todos os sinais estão disponíveis. Entretanto, o escopo temporal é limitado a uma unidade.

Quando a validação ocorre em um ambiente determinístico, a repetição de um mesmo teste produz os mesmos resultados. Em um ambiente real, no entanto, falta determinismo, repetibilidade e controlabilidade. Domínios de *clock* assíncronos, condições elétricas e condições do ambiente contribuem para um resultado inconsistente cada vez que um teste executa (DEORIO; KHUDIA; BERTACCO, 2011; MITRA; SESHIA; NICOLICI, 2010). Sendo assim, se uma validação com *Scan Chains* ocorresse em um ambiente determinista, seria possível iniciar a depuração em um dado ponto, restaurando o valor dos registradores, e verificando o ponto da falha do CID. Porém, na maioria da vezes, esse

não é o caso, e determinar um ponto de restauração é muito difícil pois pouco se sabe a respeito da falha que ocorreu (NICOLICI; KO, 2009).

A técnica de *Scan Chains* exige ainda uma estrutura de depuração composta por um mecanismo de disparo para iniciar a captura, um mecanismo para parar a captura, um mecanismo de configuração do *Scan Chains* e por fim um mecanismo para acessar/-capturar as informações dos registradores.

2.3.2 Técnica *Trace Buffer*

A técnica de *Trace Buffer* supera a limitação do *Scan Chains* ao possibilitar observar sinais por um determinado período de tempo. A Figura 4 apresenta um diagrama de bloco para depuração pós-silício utilizando a técnica *Trace Buffer*. A técnica *Trace Buffer* utiliza uma **memória intermediária** (*Trace Buffer*) para armazenar os valores monitorados. O módulo de **lógica de disparo** monitora o comportamento do CID e inicia a captura dos dados quando condições predefinidas são atingidas. A **rede de interconexão** é responsável por selecionar quais sinais serão armazenados na memória intermediária.

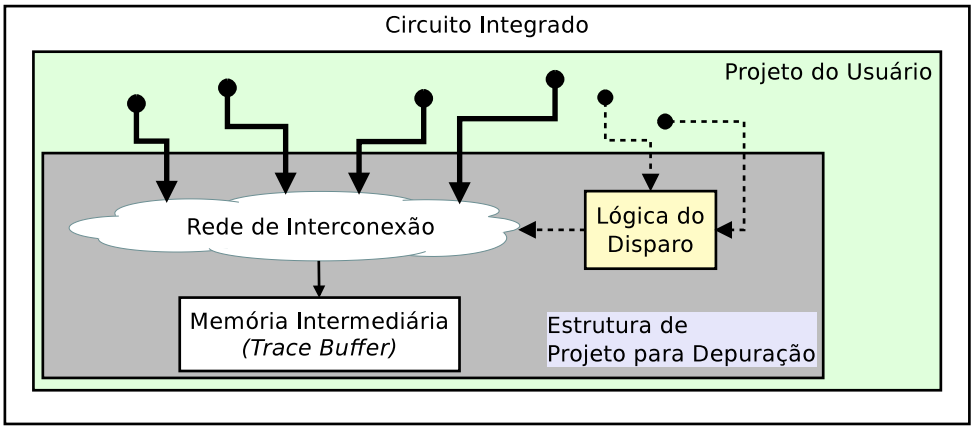


Figura 4 – Método *Trace Buffer*. Diagrama de blocos de uma infraestrutura de depuração pós-silício com *Trace Buffer*.

O tamanho da memória intermediária é definida por dois fatores: **largura** e **profundidade**. A quantidade de sinais monitorados determina a largura, e a profundidade é determinada pela quantidade de vezes que um determinado sinal é armazenado. Se considerarmos uma memória intermediária de 16 Kbits, é possível explorar diferentes configurações. Por exemplo, é possível utilizar uma determinada memória intermediária para armazenar 16 sinais durante 1024 períodos de tempo, ou utilizar o mesmo espaço para armazenar 64 sinais durante 256 períodos de tempo. A natureza do erro sendo depurado deve ser levada em consideração para definir o melhor compromisso entre largura e profundidade.

O *Trace Buffer* é uma técnica que tem um escopo espacial bem limitado, no entanto, o escopo temporal é maior em relação à técnica de *Scan-Chains* e ambas as técnicas

apresentam a desvantagem da necessidade de adição de hardware extra na depuração, aumentando os custos.

Como somente alguns sinais podem ser incluídos na estrutura de depuração, a seleção dos sinais monitorados é um passo fundamental no método de *Trace Buffer*. A seleção de sinais pode ser realizada por um processo automático ou manual. No processo manual a decisão de quais sinais incluir na depuração é baseada no conhecimento dos projetistas sobre quais módulos ou áreas do circuito apresentam maior risco no projeto (ABRAMOVICI et al., 2006). No processo automático os sinais são analisados para definir o conjunto de sinais que será conectado à rede de interconexão. A análise dos sinais pode ser estática, na qual apenas a definição dos módulos é avaliada ou dinâmica, que leva em consideração o comportamento do circuito.

2.3.2.1 Seleção manual de sinais

Falhas provenientes de Máquinas de Estados Finitos (MEFs) são mais comuns do que falhas em caminhos de dados. Novos módulos de Propriedades Intelectuais (PIs) apresentam um alto risco. Pois ainda não foram plenamente testados em conjunto com outros módulos. Uma nova lógica implementada também oferece um alto risco se não tiver sido bem testada no pré-silício.

Para definir se um módulo foi bem testado são utilizados critérios de cobertura de código (SANGUINETTI; ZHANG, 2010; BALSTON et al., 2013) e cobertura de asserções (PAL et al., 2008; ATHAVALE et al., 2014). Se um determinado módulo foi testado e os índices de cobertura de código e asserções são altos, pode ser considerado que esse módulo tem baixo risco. Se um módulo foi verificado por verificação formal, o módulo tem baixíssimo risco (ABRAMOVICI et al., 2006). No entanto, não há garantias de que não haverá erros provenientes desses módulos. Pode haver ainda problema de temporização, além de transformações durante o processo de desenvolvimento que poderiam causar alguma falha elétrica.

Outros sinais para incluir são sinais de entrada para asserções. Esses sinais podem ajudar na detecção de outras falhas. Sinais de controle de eventos e transações também são importantes para identificar falhas.

2.3.2.2 Seleção automática de sinais

A seleção manual é processo que requer conhecimento do projeto do CID e é uma tarefa que demanda bastante tempo para ser completada (KO; NICOLICI, 2009). Sendo assim, alguns métodos foram propostos para automatizar a seleção dos sinais (KO; NICOLICI, 2009; LIU; XU, 2012b). Essa seleção automática de sinais pode ser baseada no fato de que a inclusão de determinados sinais ajuda a restaurar o valor de outros sinais não capturados. Assim, pode-se reduzir o tempo gasto para completar a tarefa e ainda

beneficiar o processo, encontrando relacionamentos de sinais que dispensa a inclusão de alguns sinais e permite a inclusão de outros.

Um modo de selecionar os sinais para depuração, é incluir sinais com maior restaurabilidade de estados. A restauração de estados é uma técnica que utiliza os sinais capturados para restaurar valores de outros sinais em estados presente, passado e futuro (HUNG; WILTON, 2013). Essa técnica de restauração busca aumentar a quantidade de informação disponível, porém não considera a real utilidade dessa informação. Isto é, um sinal com alta correlação com outro sinal capturado, pode não prover qualquer informação útil durante o processo de depuração. Considere por exemplo, informações provenientes de *pipeline*, que são triviais e podem não ser úteis no processo de depuração.

Trabalhos apresentados por Yang et al. (2010) e Prabhakar e Hsiao (2009) modelam o problema de seleção de sinais como métodos formais de Problemas de Satisfazibilidade Booleana (SATs) para determinar quais sinais terão mais eficiência na restauração dos sinais não observados. Yang e Touba (2009) apresentam um método de programação linear inteira para escolher os sinais com maior sensibilidade à erros injetados e diminuir o tempo entre a ocorrência do erro e sua detecção.

Independente do método de seleção de sinais escolhido pelo projetista de hardware, o bloqueio de alguns sinais na rede interconexão, diminui a observabilidade do circuito e capacidade de restaurabilidade no caso de um método automático. Na seção 2.4 é apresentada a contextualização sobre redes de interconexão (comutação de circuitos). Essa contextualização é importante para a apresentação da rede Ômega assimétrica, que é a rede proposta nesse trabalho para substituição das redes Mux Trees em um Projeto para Depuração (PpD). A rede Ômega assimétrica é uma rede capaz de diminuir a taxa de bloqueio e aumentar a observabilidade do circuito em depuração.

2.4 Redes de Interconexão

Redes de interconexão são sistemas programáveis utilizados para comunicação entre dois componentes. Sua função é transportar os dados. A Figura 5 apresenta uma rede de interconexão hipotética. Na Figura 5, cada componente é identificado com uma letra de A até D. Para o componente A enviar uma mensagem para D, A envia junto com a mensagem a informação sobre o destino. A rede de interconexão avalia a requisição e direciona para o destinatário. A rede de interconexão é um sistema porque possui diversos componentes como memórias intermediárias, nodos de roteamento (*switches*), controles, e canais de dados. A rede de interconexão é programável porque realiza diferentes conexão no tempo (DALLY; TOWLES, 2003).

A função de transporte de dados de uma rede de interconexão ocorre em três níveis: nível de rede no chip (SoC), nível de sistema, e em nível de área local (ou de área

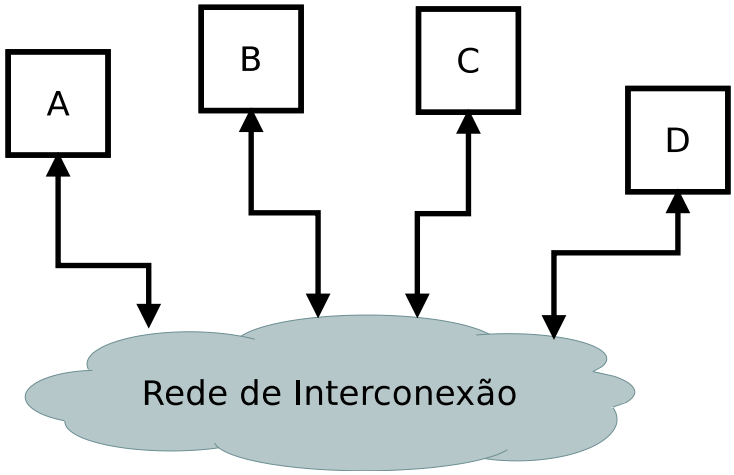


Figura 5 – Rede de Interconexão hipotética. Funcionamento de uma rede de interconexão com canais bidirecionais.

ampla). No nível de *chip*, as redes são utilizadas para interconectar vetores de memória e registradores dentro de um chip específico, como um processador aos bancos de memória e banco de registradores. Em nível de sistema, é utilizado para conectar memórias e processadores, ou portas de entradas e saídas. Em nível local ou amplo, conecta-se sistemas dentro de uma organização ou no mundo. Nesse trabalho podemos considerar que é a rede proposta está no nível de *chip*/SoC.

A utilização de redes de interconexão é ampla. Está presente em quase todos os sistemas que precisam interconectar mais de dois componentes. São utilizados em *switches* de comunicação, como telefone e sistemas de computador. Nos sistemas de computador conectam dispositivos como processadores e memórias, dispositivos de entrada e saída e controladores de entrada e saída. As redes de interconexão possuem um papel bastante importante dentro de um projeto, pois a rede limita o desempenho de todo sistema. Isto pode ocorrer pela rede determinar a latência máxima do circuito, a quantidade de banda ou volume de dados máximo que pode trafegar na rede. Outro fator que limita o desempenho e o projeto é a quantidade máxima de portas de entrada e saída.

Ao implementar ou escolher uma rede de interconexão para uma determinada aplicação o projetista deve lidar com especificações de desempenho e restrições de tecnologia para lidar com a topologia e roteamento da rede. O ponto chave da utilização de uma rede de interconexão é o compartilhamento de recursos, incluindo os caminhos compartilhados e nós de roteamento. O padrão de conexão entre os *switches* é chamado de topologia. A mensagem a ser entrega pode passar por vários *switches* e vias até chegar ao destino, e o roteamento da rede é que determina os possíveis caminhos para entregar os dados.

Na depuração pós-silício, a rede de interconexão precisa ser reconfigurável, o que torna possível para o projetista executar vários testes no dispositivo utilizando a mesma infraestrutura. Para construir uma rede com este comportamento é necessário adicionar

registradores que podem habilitar a reconfiguração da rede.

Na Figura 6 é apresentado um exemplo de uma rede de interconexão com um padrão de conexão denominado Cube (PEASE, 1977). A rede possui 8 entradas e 8 saídas com 12 nós. Cada *switch* possui duas vias de entrada e duas vias de saída. Para conectar a entrada 000_b à saída 101_b nessa rede, apenas a rota destacada é possível.

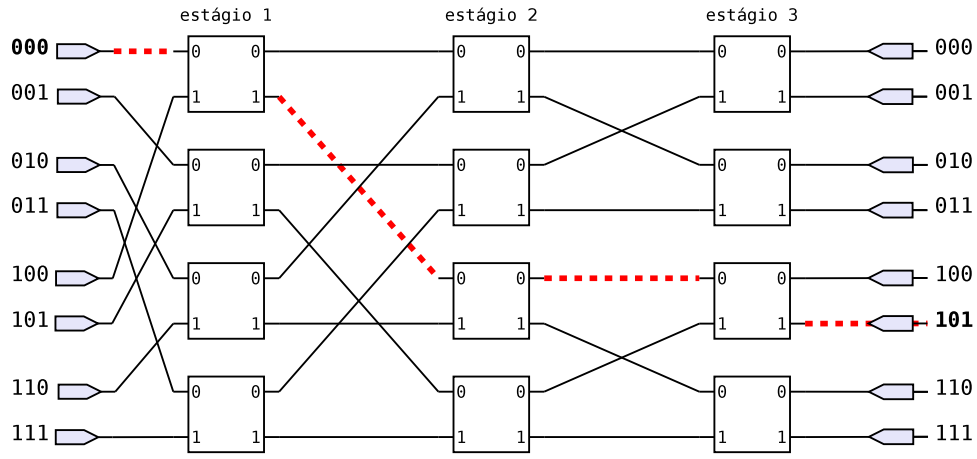


Figura 6 – Exemplo de rede com topologia Cube.

2.4.1 Redes não bloqueantes

Nas seções a seguir são apresentadas as redes Crossbar e Clos (CLOS, 1953). Nesse trabalho são contextualizadas apenas as redes de comutação de circuito, que são caracterizadas pela configuração da rota antes do início da comunicação. A rede Crossbar é o nó de roteamento base de outras redes e por isso é discutida nesse trabalho. Também é apresentada a rede Clos, uma rede com características não bloqueante como a Crossbar, porém que requer menos recurso.

Em uma rede não bloqueante, uma requisição de conexão sempre é bem sucedida se o destinatário não estiver ocupado. Assim, uma rede não bloqueante tem capacidade de permutação das entradas e saídas. Para isto, um caminho dedicado é estabelecido entre a entrada e a saída, sem compartilhamento de canal. Já em uma rede bloqueante, não é possível fazer todas as conexões sem que haja conflito.

Uma rede não bloqueante pode ser **não bloqueante rearranjável** ou **estritamente não bloqueante**. Em uma rede estritamente não bloqueante, é possível estabelecer todas as conexões sem que haja necessidade de rearranjar as conexões já estabelecidas. Já nas redes rearranjáveis, também é possível qualquer permutação, porém pode ser necessário alterar a rota de uma ou mais conexões para realizar uma determinada conexão.

2.4.1.1 Redes Crossbar

Uma Crossbar é uma rede de estágio único que conecta n entradas à m saídas utilizando *switches crosspoints*. Conexões *singlecast* e *multicast* são possíveis. No modo *singlecast*, um arbitro garante que somente uma entrada é conectada à uma única saída. Uma rede Crossbar assimétrica é apresentada na Figura 7. Uma rede Crossbar requer $n.m$ elementos de *crosspoint* (GRAMMATIKAKIS; HSU; KRAETZL, 2000).

Na Figura 7 não há distinção entre entrada e saída. Os canais são bidirecionais, semelhante aos primeiros modelos de Crossbar utilizados na telefonia. Para esta representação de um Crossbar de $4x2$, são utilizados 8 *crosspoints*.

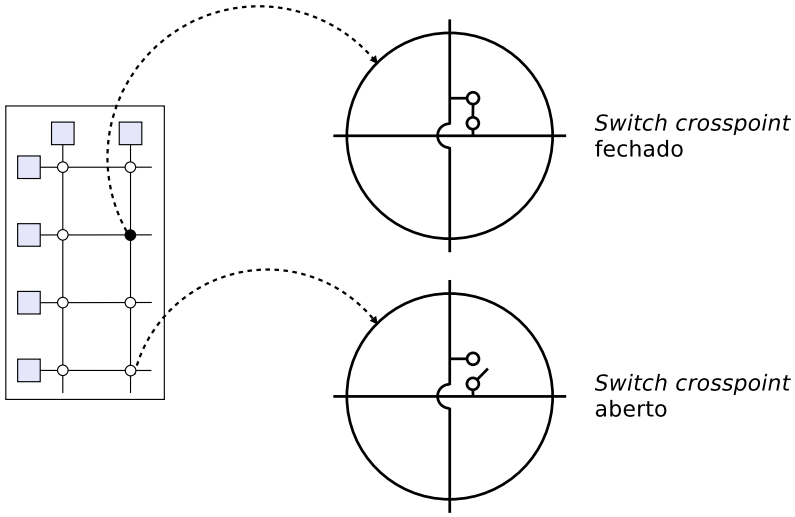
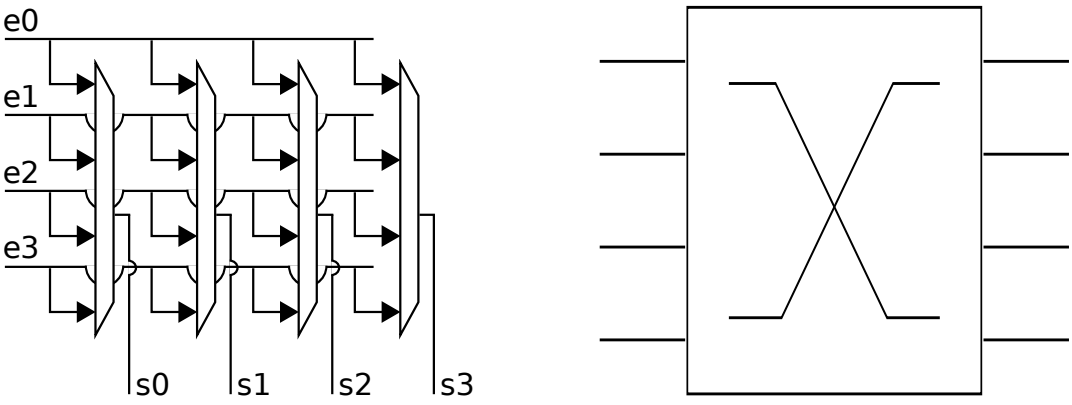


Figura 7 – Crossbar 4 x 2 com 8 *switches crosspoints*.



(a) 4 x 4 Crossbar digital. (b) 4 x 4 Crossbar simplificada.

Figura 8 – Representações da rede Crossbar. Em (a) uma Crossbar com 4 *switches* com multiplexadores 4x1. Todas as entradas da Crossbar, {e0, e1, e2, e3}, são conectadas aos *switches*. E as saídas dos *switches*, {s0, s1, s2, s3}, são conectadas às saídas da Crossbar. Em (b) a representação da rede Crossbar de forma simplificada.

Em lógica digital a implementação de um Crossbar é semelhante ao apresentado na Figura 8a. Cada *switch* consiste de m multiplexadores de tamanho $nx1$. São n entradas em cada *switch* e a saída do *switch* é conectada à porta de saída da Crossbar. Para facilitar, o desenho de uma Crossbar dentro de um sistema, pode-se omitir detalhes de *switches* e multiplexadores. A representação simplificado é um **X** dentro de um retângulo conforme ilustra a Figura 8b

2.4.1.2 Redes de Interconexão Multiestágio

Uma rede Crossbar apresenta um custo quadrático, que é considerado alto para muitas aplicações, incluindo depuração pós-silício. Na depuração pós-silício, é comum ter um número muito grande de sinais a serem observados, e poucos deles podem ser rastreados ao mesmo tempo. Se for considerado um Crossbar de $4096x32$, seriam gastos 32 multiplexadores de tamanho $4096x1$.

Redes Rede de Interconexão Multiestágios (RIMs) reorganizam os *crosspoints* em diversos *switches* que geralmente são redes Crossbares. Os *switches* são então organizados em estágios. Na Figura 9 é apresentado uma rede RIM hipotética. Nessa figura, os sinais de entrada da rede são distribuídos nos diversos Crossbar do primeiro estágio. Os sinais podem passar por diversos estágios e caminhos até o último estágio. Na Figura 9, a rede possui N entradas, M saídas e S estágios. A conectividade completa é atingida com $O(logN)$ estágios, considerando N como o número de entradas, e $N > M$, sendo M o número de saídas. RIMs também podem ser categorizadas por bloqueantes, não bloqueantes e rearranjáveis.

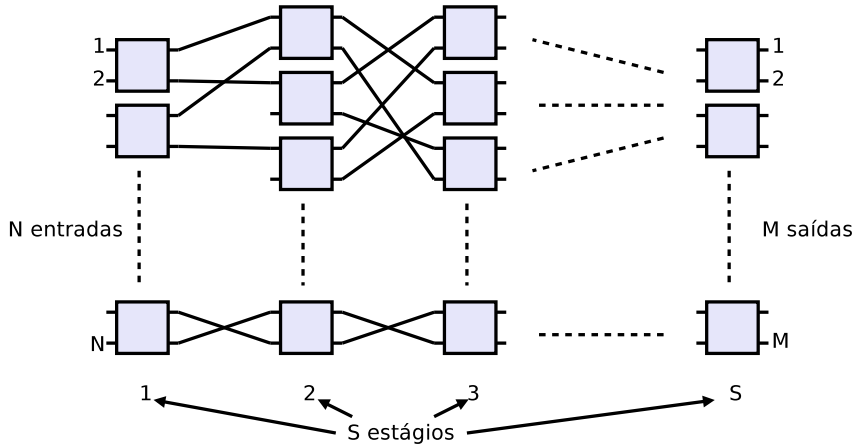
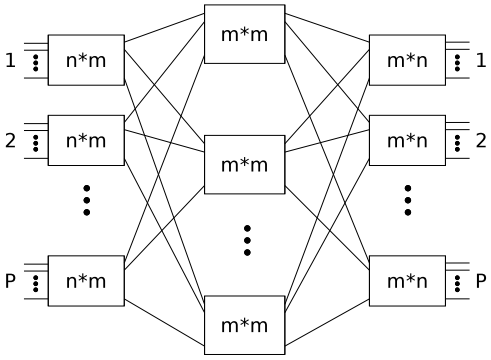


Figura 9 – Rede de interconexão multiestágio com N entradas, M saídas e S estágios.

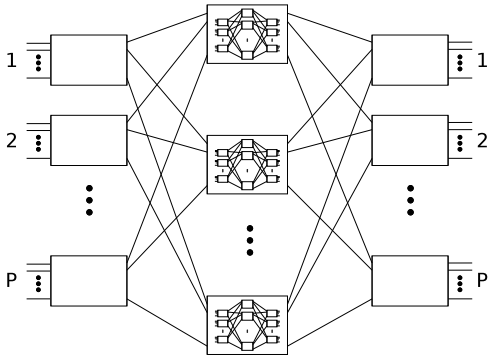
2.4.1.3 Redes Clos

As redes Clos foram projetadas como uma RIM não bloqueante e uma alternativa as redes Crossbar (CLOS, 1953). Uma rede Clos de 3 estágios $C(n, m, p)$ é composta por

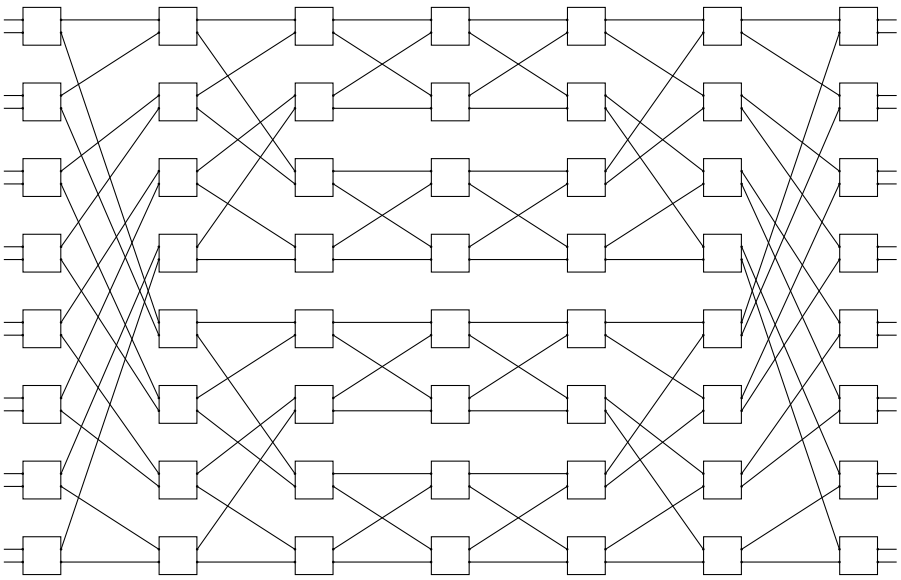
um estágio de entrada, um estágio intermediário, e um estágio de saída. Seja n, o número de entradas por *switch*, p o número de *switches* no primeiro estágio, e m o número saídas no primeiro estágio. O estágio intermediário tem *switches* de tamanho $m*m$. O estágio de saída possui p *switches* de tamanho $m*n$. O número de *switches* no estágio intermediário para que não haja bloqueio é $2n-1$. A construção de uma rede Clos de 3 estágios é apresentada na Figura 10a.



(a) Rede Clos de 3 estágios.



(b) Rede Clos de 5 estágios.



(c) Rede Benes 16x16 com 7 estágios.

Figura 10 – Rede Clos e Benes.

Para rotear um caminho em uma Clos de 3 estágios estritamente não bloqueante, basta escolher uma rota para um *switch* intermediário livre. Isto porque uma das propriedades da Clos é que há caminhos para todos os *switches* intermediários provenientes do primeiro ou terceiro estágio.

Uma rede Clos multiestágios é composta por camadas. Seja s um número inteiro maior que um. Uma rede Clos de $(2*s-1)$ estágios possui n^{s+1} entradas, e p *switches* de tamanho $n*m$ no primeiro estágio. No estágio intermediário, m redes Clos de $(2*s-1)$

estágios. No terceiro estágio, p *switches* de tamanho $m * n$. A Figura 10b apresenta a construção recursiva da rede multiestágio.

Seja n_1 o número de entradas dos *switches* do primeiro estágio, e n_2 o número de entradas dos *swtiches* do terceiro estágio. Uma rede não bloqueante Clos com 3 estágios pode ser construída com o mínimo de $(n_1 - 1) + (n_2 - 1) + 1$ *switches* no estágio intermediário. Tanto a rede Clos simétrica como a assimétrica podem ser geradas como uma RIM rearranjável. Uma rede Clos é rearranjável se e somente se $m \geq n$. Uma rede Clos assimétrica é rearranjável se e somente se $m \geq max\{n_1, n_2\}$ (SLEPIAN, 1952; DUGUID, 1959; HWANG, 2004).

O roteamento em uma rede Clos rearranjável pode ser feita utilizando um grafo bipartido. Os *switches* de entrada representam um dos conjuntos do grafo, e os *switches* de saída o outro conjunto. Assim, o problema de roteamento se torna um problema de coloração em grafos. O roteamento também pode ser feito utilizando decomposição de matrizes e pode ser aplicado tanto para Clos estritamente não bloqueante, quanto para Clos rearranjável (DALLY; TOWLES, 2003).

A Figura 10c apresenta uma rede Benes 16x16. Esta rede é um caso especial da rede Clos composta somente por *switches* de tamanho 2x2. A rede Benes requer o número mínimo de *switches crosspoints* para construir uma rede rearranjável não bloqueante.

2.4.2 Redes Bloqueantes

Redes bloqueantes podem ser classificadas em: **Banyan** (GOKE; LIPOVSKI, 1973), **Delta** e **Bidelta** (PATEL, 1979). Essas redes não possuem a quantidade de *switches* necessários para permitir qualquer combinação de entrada e saída ao mesmo tempo. Existe uma grande variedade de redes bloqueantes, sendo que algumas delas são topologicamente equivalentes. Essas redes são chamadas de isomórficas. Dois procedimentos para verificar se duas redes são isomórficas são apresentadas em (ORUC; ORUC, 1985) e (SRIDHAR, 1989).

Redes Banyan são uma classe de redes que possuem um único caminho de um nó de entrada até um nó de saída. Para ser uma rede Banyan também é necessário que haja acesso total à rede, isto é, deve ser possível conectar qualquer nó de entrada em qualquer nó de saída da rede. Redes Banyan geralmente apresentam *switches* de tamanho $k * k$ e que são organizados em estágios, sendo k um inteiro maior que 1. Na Figura 11 é apresentada uma rede Banyan. A Figura 12 não é uma rede Banyan pois não é possível conectar o nó de entrada 010_b nos nós de saída 110_b ou 111_b.

Uma rede para ser caracterizada como Delta, precisa que duas condições sejam atendidas. A primeira condição para ser uma rede Delta é que a rede seja Banyan. A segunda condição que caracteriza a rede Delta é que toda configuração dos *switches* é

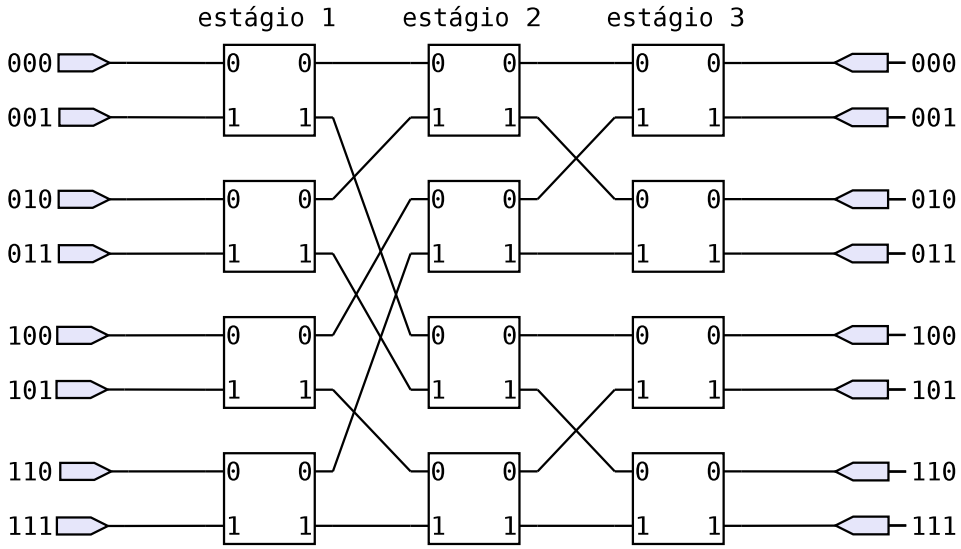


Figura 11 – Rede Banyan. Essa rede apresenta um padrão de conexão denominado Baseline.

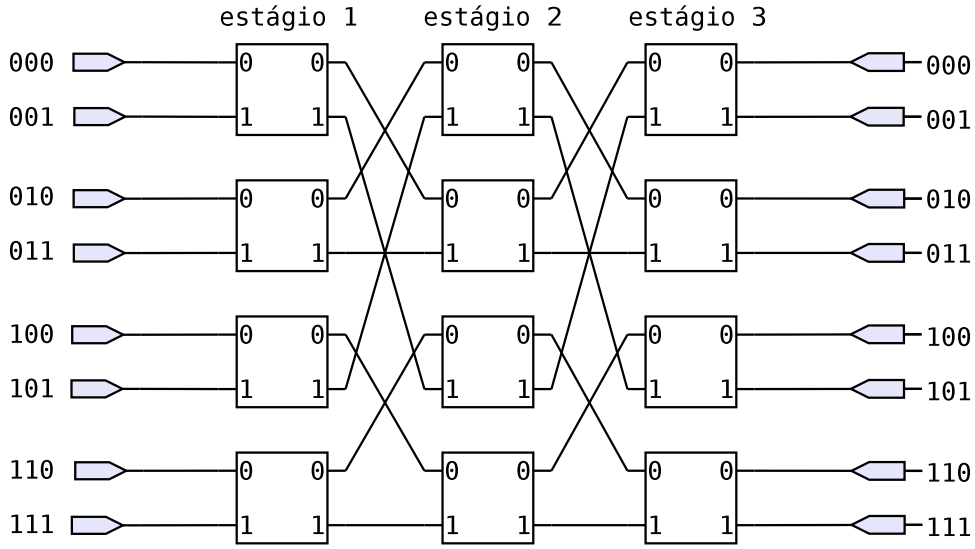


Figura 12 – Rede não Banyan. A metade de cima da rede está conectada a parte de baixo por apenas por um caminho, e não é possível conectar qualquer entrada em qualquer saída. Por exemplo, não é possível conectar a entrada 110_b na saída 011_b.

determinada de forma única, baseada no rótulo do destino e no estágio do switch. Se as propriedades da rede Delta são preservadas quando a rede é invertida, a rede é chamada de Bidelta. Na Figura 13 é apresentada uma rede não Delta. Essa rede não é Delta porque no último estágio não é mantido o padrão de conectividade. Na Figura 14 é apresentado uma rede que não é Bidelta, pois, o rótulo de roteamento não é o mesmo quando a rede é invertida (SIBAL; ZHANG, 1995). Na Figura 15 é apresentada uma rede Bidelta.

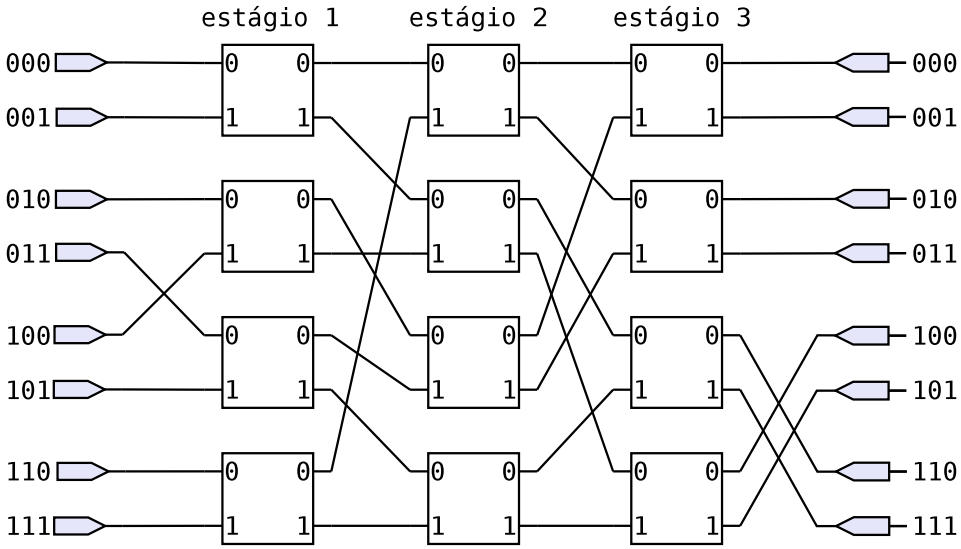


Figura 13 – Rede não Delta. Essa rede não é uma rede delta pois o rótulo de destino 100_b não determina o caminho para o nó de destino correspondente.

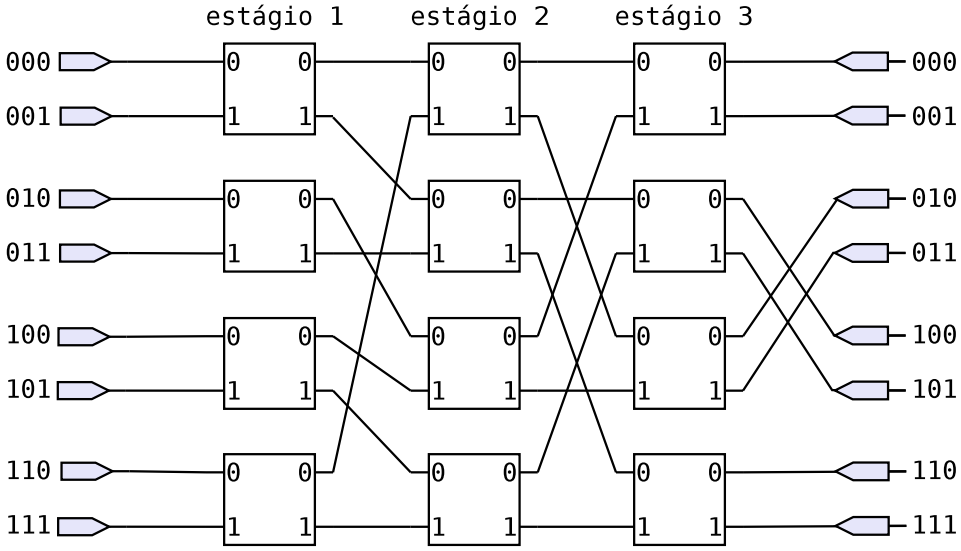


Figura 14 – Essa rede é Delta, mas não é Bidelta pois ao invertê-la não é possível fazer o roteamento por rótulo do destino. Por exemplo, a rota inversa a partir do nó 000_b utilizando o rótulo 011_b, leva ao nó 111_b.

2.4.2.1 Redes Ômega

A rede Ômega (LAWRIE, 1975) é uma rede da família **k-ary n-flies** (Butterflies) (DALLY; TOWLES, 2003). Em uma rede k-ary n-flies, k define o tamanho do switch (**radix**), e n define o número de **estágios**. Dentre as redes que compõem a família de redes k-ary n-flies, estão as redes Butterfly, Ômega, Cube, Flip, Baseline e outras. Essas redes são isomórficas, e a distinção se dá apenas pelo padrão de conexão entre estágios (WU; FENG, 1980). Na Figura 15 é apresentado uma rede Butterfly 8x8 (2-ary 3-flies). Na Figura 16 é apresentado uma rede Ômega 8x8 (2-ary 3-flies).

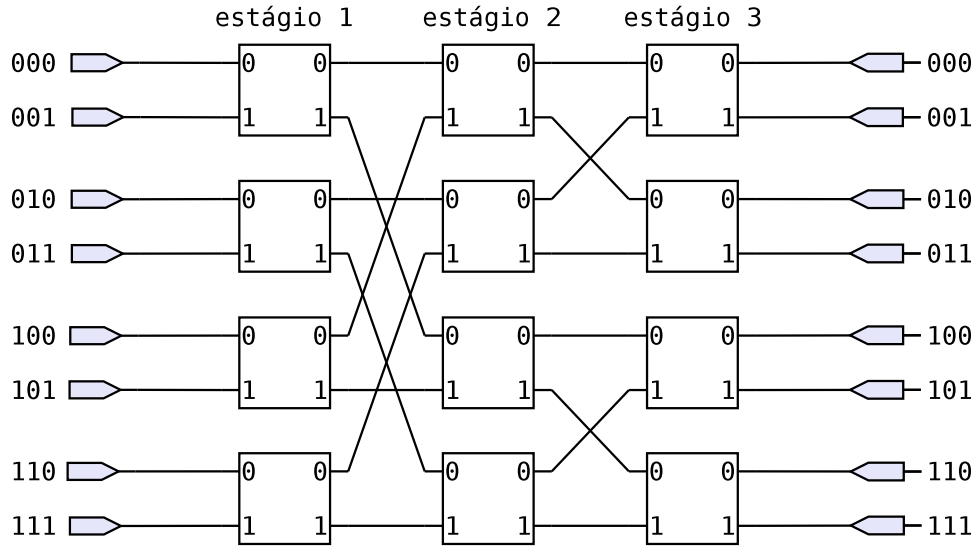


Figura 15 – Essa rede é Bidelta, pois apresenta as características de rede Banyan e o roteamento normal/inverso pode ser feito com base no rótulo do destino. Por exemplo, se invertida, é possível chegar ao nó 100_b utilizando o rótulo 100_b para configurar os *switches*, e pode ser feito a partir de qualquer nó de origem da rede. O padrão de interconexão dessa rede dá a ela o nome de Butterfly.

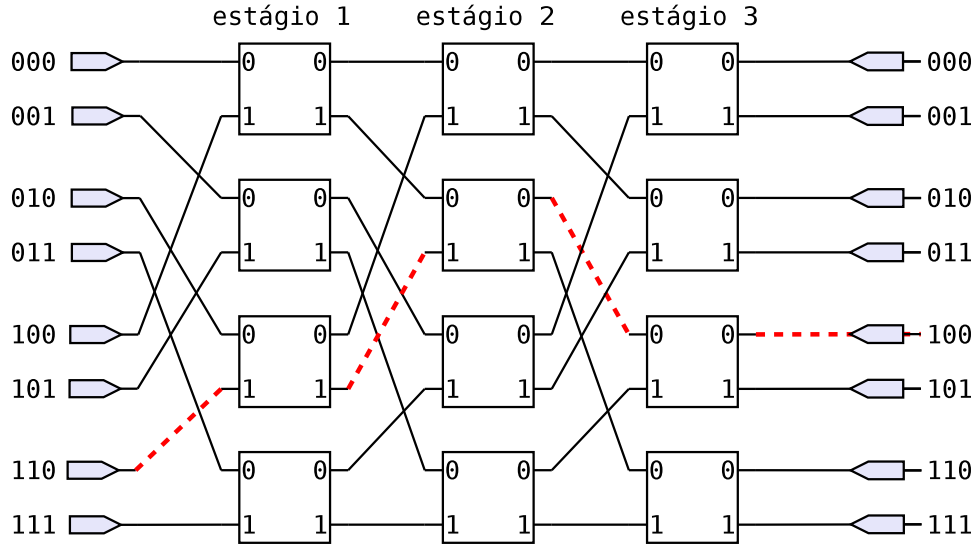


Figura 16 – Rede Ômega 8x8. A rede Ômega é uma rede Bidelta e também compõe a família k-ary n-flies.

A rede Ômega é uma extensão da rede **Shuffle-Exchange** (STONE, 1971). Uma rede Shuffle-Exchange utiliza o padrão de interconexão **Perfect-Shuffle**. Muitas aplicações utilizadas em computação paralela utilizam este padrão de interconexão. Como o padrão de interconexão Perfect-Shuffle é fixo para qualquer estágio, a rede Shuffle-Exchange pode ser construída de forma cíclica com apenas um estágio, conforme é apresentado na Figura 17. A capacidade de realizar qualquer permutação é atingida com $2logN - 1$ ciclos, sendo N o número de entradas/saídas da rede. Essa rede no entanto, não pode ser

utilizada para comutação de circuitos sem que haja a inclusão de memórias intermediárias (GRAMMATIKAKIS; HSU; KRAETZL, 2000). A rede Ômega remove ciclos e inclui $log_2 N$ estágios. Assim, ela apresenta a característica de uma rede Banyan, com uma única rota para um par de entrada e saída. A Rede Ômega em sua concepção original também é uma rede Bidelta.

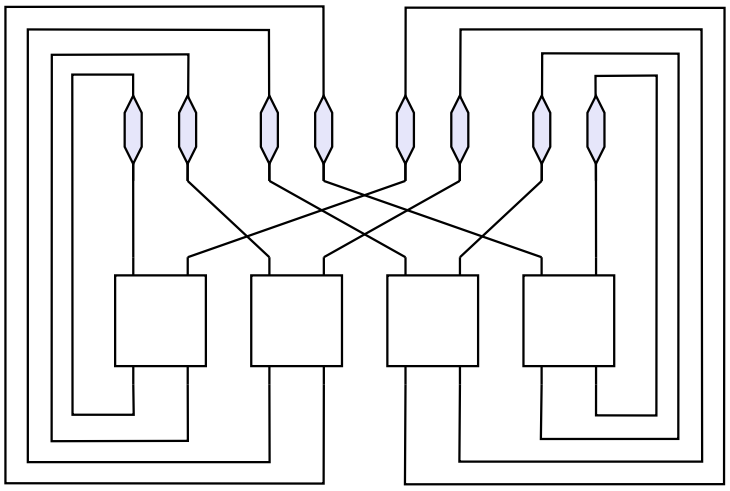


Figura 17 – Rede Shuffle-Exchange 8x8. Rede cíclica com padrão de interconexão Perfect-Shuffle. Realiza qualquer permutação com $2logN - 1$ ciclos.

Fonte: (GRAMMATIKAKIS; HSU; KRAETZL, 2000).

Essa família de redes possui canais unidirecionais e o fluxo geralmente é representado da esquerda (entrada) para direita (saída). Em uma rede k-ary n-flies, o número de conexões para outros nós, ou grau, é $2k$. Logo, em uma rede com radix-2, um *switch* possui 4 conexões para outros *switches*. O maior caminho calculado entre todos os menores caminhos, ou o diâmetro da rede, é $log_k(N) + 1$, sendo N o número de entradas/saídas. Assim, para uma rede com 8 entradas e radix-2, o diâmetro da rede é $log_2(8) + 1 = 4$.

Redes k-ary n-flies possuem k^n terminais no estágio de entrada. Cada um dos n estágios possui k^{n-1} Crossbares de tamanho $k * k$. O estágio de saída é semelhante ao estágio de entrada e possui k^n terminais. Se agrupados os pares de terminais de entrada e saída, o total de terminais da rede é $N = k^n$.

As características de diâmetro e grau das redes k-ary n-flies apresentam uma desvantagem: apesar deste valor ótimo, as redes não possuem diversidade de caminhos na concepção original. Portanto, só há um caminho disponível de uma entrada para uma saída da rede. Para aumentar a diversidade de caminhos, é possível incluir estágios extras (SIEGEL, 1979; FERREIRA; VENDRAMINI; NACIF, 2011). Na Figura 16 é apresentado o roteamento de $110_b \rightarrow 100_b$ em uma rede Ômega convencional, sem adição de estágios extras. A Figura 18 também apresenta uma rede Ômega, porém com um estágio extra. É possível observar que para conectar o mesmo par de entrada/saída, duas rotas são possí-

veis. Se outro estágio for adicionado, 4 rotas serão possíveis. Assim, a cada novo estágio adicionado, o número de rotas para conectar um par de entrada/saída dobra (FERREIRA et al., 2011).

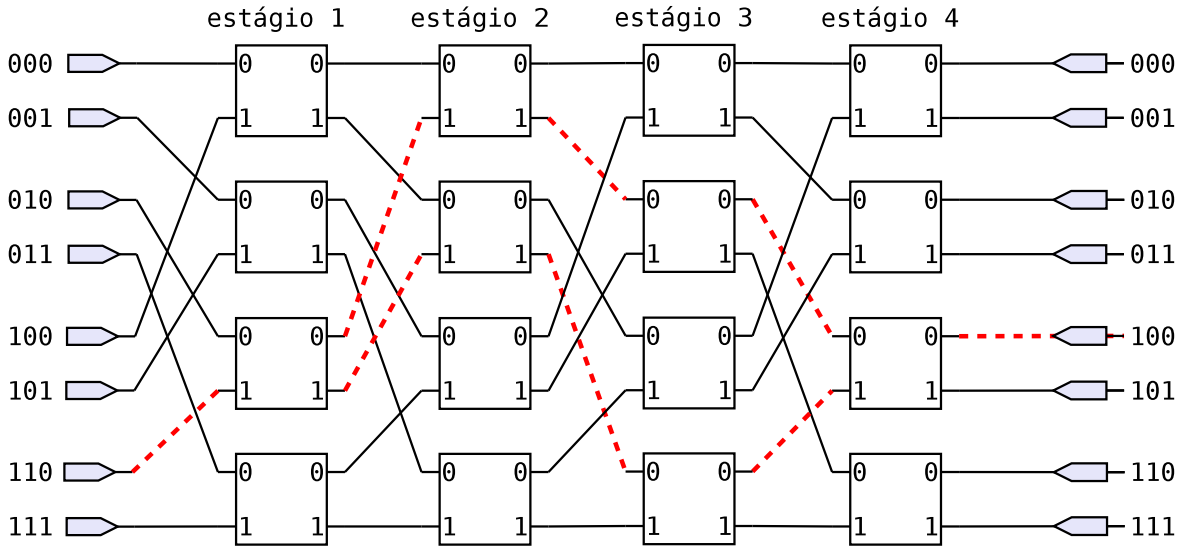


Figura 18 – Rede Ômega 8x8 com 1 estágio extra. Ao rotear a porta de entrada 110_b para a porta de saída 100_b, dois caminhos são possíveis. A regra geral é que a cada novo estágio adicionado, o número de possíveis rotas dobra.

O roteamento na rede Ômega é feito pela técnica de roteamento por rótulo do destino. O rótulo do destino em binário é interpretado como a rota. Essa técnica também é conhecida por auto roteamento. Em uma rede por comutação de pacotes, a rota do pacote estaria no cabeçalho do pacote. No contexto de comutação de circuito, o roteamento é realizado antes da execução. Este tipo de roteamento é bastante útil quando se conhece o comportamento do tráfego (GRAMMATIKAKIS; HSU; KRAETZL, 2000 apud LAHAUT; GERMAIN, 1994, p. 59–60), como é o caso desse trabalho. No exemplo apresentado na Figura 16, a porta de entrada 110_b é conectada à porta de saída 100_b. Para estabelecer essa conexão utilizando o método de roteamento pelo rótulo do destino, cada bit indica o roteamento de um *switch*. O primeiro bit do destino é 1, logo, no primeiro estágio, a saída do *switch* deverá ser pela porta inferior. O segundo e o terceiro bits são 0, logo, os respectivos *switches* do estágio 2 e 3 deverão ser configurados para que a saída seja pela porta superior.

Conforme apresenta a Figura 19, duas operações são possíveis na rede Ômega com radix-2: direto e cruzado. Um segundo método de roteamento para a rede Ômega é chamado de roteamento por XOR dos rótulos. Para exemplificar, observe novamente a Figura 16. A operação de roteamento resulta em $110_b \oplus 100_b = 010_b$. Cada bit do resultado da operação XOR determina uma dessas operações em um determinado estágio, sendo o bit da esquerda mais significante utilizado para o primeiro estágio. Assim, se o bit é 0, então a operação direta é utilizado. Se o bit é 1, então a operação é cruzado. No exemplo

apresentado, 010_b indica que o roteamento será direto, cruzado, direto. Para redes com outros tamanhos de *radix* são utilizados $log_2 d$ bits para determinar a saída do *switch*, sendo d o grau do *switch*. O tamanho do rótulo é dado por $Dlog_2 d$, sendo D o diâmetro da rede. Redes Ômega com estágios extras podem ser roteadas utilizando um algoritmo de janelas conforme é apresentado em (VENDRAMINI; FERREIRA, 2010).

(a) Direto (b) Cruzado

Figura 19 – Conexões disponíveis nos *switches* Ômega

2.4.2.2 Rede de Árvores de Multiplexadores Encadeados

Devido ao baixo custo de área, na indústria, redes Mux Trees são utilizadas como uma solução padrão para interconectar os sinais no processo de validação (ABRAMOVICI, 2008). Normalmente essa rede é construída por multiplexadores do mesmo tamanho, geralmente multiplexadores 2x1, que são conectados em estágios. O número de estágios para as redes Mux Trees compostas por multiplexadores 2x1 é $\log_2(input) - \log_2(output)$. As redes Mux Trees são bloqueantes e existe apenas um caminho entre uma entrada e uma saída da rede. Essas características minimizam o número de estágios e *switches* da rede. A Figura 20 apresenta uma rede Mux Tree 8x2 conectada à uma memória intermediária. A porta de entrada 0 está conectada à saída 0; e a porta de entrada 5 está conectada à saída 1. A seleção da porta 5 faz que seja impossível rastrear concorrentemente qualquer uma das portas do conjunto {4, 6, 7}.

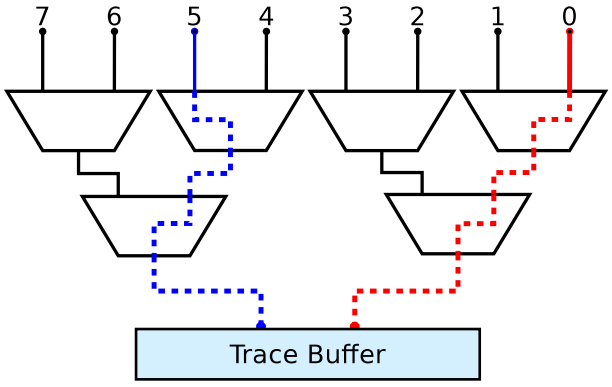


Figura 20 – Rede Mux Tree 8x2

Considerando uma rede com 8 entradas e 2 saídas, uma rede Mux Tree requer 2 estágios. Uma rede Clos de 3 estágios requer somente um estágio adicional, contudo utiliza mais multiplexadores. Redes Clos também utilizam multiplexadores de tamanhos

variados. Uma rede Benes requer $2log_2N - 1 = 5$ estágios. Considerando uma rede de 1024x32, uma rede Mux Tree requer 5 estágios, e uma rede Benes requer 19 estágios. Se a saída da rede é expandida para 64 saídas, o número de estágios em uma rede Benes ainda será 19, porém em uma rede Mux Tree o número de estágios é reduzido para 4.

Quando uma rota não pode ser estabelecida entre uma entrada e uma saída, um bloqueio ocorre. Neste trabalho é considerado que a métrica taxa de bloqueio mede a **quantidade de sinais roteados** sobre a **quantidade de saídas disponíveis da rede**. Para exemplificar, considerando a rede apresentada na Figura 20, e que o conjunto de entrada é composto por {1, 3}, a taxa de bloqueio para este conjunto é 50%. Essa métrica foi estabelecida com o seguinte princípio: - existe um conjunto de sinais que trará a melhor observabilidade do circuito, seja para o entendimento do erro em si, ou para restaurar o valor de outros sinais quando se utiliza algoritmos de restauração de sinais. O inclusão de sinais fora do conjunto principal escolhido pelo projetista/algoritmo pode não contribuir para o entendimento e correção do erro.

A principal característica das rede Mux Trees é o baixo custo de área. Em contrapartida, a taxa de bloqueio para essa rede de interconexão atinge em média 36% para combinações aleatórias. RIMs não bloqueantes, como redes Clos por exemplo, apresentam um custo de área proibitivos para serem inseridos dentro de uma infraestrutura de PpD. Neste trabalho é proposto a substituição da rede Mux Tree dentro de uma infraestrutura de PpD por uma rede Ômega. A rede Ômega é bastante utilizada em muitas aplicações, mas é sempre explorada como uma rede **simétrica** (DALLY; TOWLES, 2003; GRAMMATIKAKIS; HSU; KRAETZL, 2000). A seção 3 apresenta como criar uma nova rede, que é uma versão **assimétrica** da rede Ômega.

3 Método

Na depuração pós-silício, uma rede assimétrica com N entradas e M saídas, com $N > M$, conecta os sinais monitorados à memória intermediária. Nesse trabalho é proposta uma rede Ômega para ser utilizada como um dispositivo de rede de interconexão assimétrico na depuração pós-silício. Nesse capítulo, os trabalhos relacionados são apresentados na seção 3.1. Na seção 3.2 é apresentado o método para a geração da rede Ômega assimétrica proposta e uma heurística para determinar a taxa média de bloqueio para as redes Ômega assimétrica e Mux Tree. Em seguida, na seção 3.3 é apresentada a ferramenta proposta para geração de redes de interconexão para depuração pós-silício, e como gerar as redes Mux Tree, Clos multiestágio e Benes para realizar a comparação com as redes Ômega assimétricas.

3.1 Trabalhos relacionados

Narasimha (1994) apresenta um concentrador de tamanho N construído a partir de 2 concentradores de tamanho $N/2$. O concentrador é construído de forma recursiva e apresenta uma topologia equivalente à uma rede Ômega inversa. A rede Narasimha foi introduzida em um contexto no qual existem entradas ativas e inativas, sendo que a função do concentrador é rotear todas as entradas ativas. Para isso, entradas com o bit mais significante igual a 0 são roteadas na parte de cima da rede. Entradas com o bit mais significante igual a 1 são roteadas pela parte de baixo da rede. Por definição um concentrador é uma rede de interconexão assimétrica, no entanto, esse é um concentrador simétrico. Se respeitadas as propriedades da rede, não haverá bloqueio no grupo de entradas ativas.

O concentrador de Narasimha (1994) foi utilizado como base para o concentrador de Quinton e Wilton (2005). Esse concentrador foi proposto para a conexão de blocos lógicos programáveis em CIDs. No trabalho de Narasimha, o número de entradas deve ser potência de 2, já em (QUINTON; WILTON, 2005), essa restrição é removida. Nessa nova rede, o último Crossbar do estágio de entrada é removido, e os sinais são conectados diretamente em cada uma das 2 sub-redes. No entanto, essa última modificação torna o concentrador bloqueante.

Liu e Xu (2009) especificam um conjunto de 3 regras para redução de componentes da rede Narasimha, porém mantendo a característica de rede não bloqueante. A arquitetura da rede (LIU; XU, 2009) utiliza duas redes Mux Tree com conjuntos de sinais mutuamente exclusivos. Essas duas redes são ligadas no concentrador não bloqueante, que é conectado à memória do *Trace Buffer*. Segundo os autores do trabalho, sinais altamente

correlacionados devem ser monitorados juntos, enquanto os outros não. Porém, sinais não correlacionados são em muitos casos essenciais para algoritmos de restauração de sinais, enquanto sinais altamente correlacionados em muitos casos são desnecessários, pois os valores poderiam ser inferidos de outros sinais (PRABHAKAR; HSIAO, 2010).

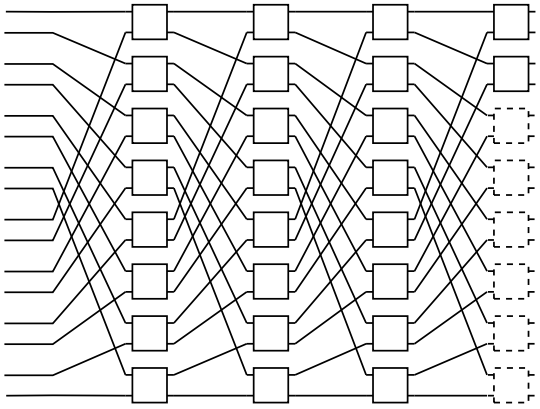
Em comparação às redes baseadas na rede Narasimha, a rede proposta nesse trabalho não considera nenhum tipo de relacionamento entre os sinais de entrada da rede. Dessa forma, a Ômega assimétrica proposta é genérica. Em contrapartida, a rede proposta apresenta casos nos quais sua aplicação é mais favorável do que em outras, como mostra o Capítulo 4. Conforme é apresentado no seção 3.2, a rede Ômega assimétrica proposta permite uma flexibilidade em relação à custo e benefício que deve ser avaliada pelas companhias e projetistas.

O trabalho de (YANG; TOUBA, 2008) apresenta uma rede XOR que agrupa um conjunto de sinais em uma única posição da memória do *Trace Buffer*. Assim, todos os sinais explorados são monitorados, porém os dados são armazenados em uma forma de compactação com perdas. Os dados armazenados na memória são denominados assinaturas de rastreio. No entanto, para localizar erros no CID, é necessário calcular as assinaturas corretas, chamadas de *golden vectores*. Ao realizar a comparação das assinaturas de rastreio com as assinaturas corretas, é possível localizar um erro em um determinado grupo de sinais rastreados. Portanto, para localizar o erro ainda é necessário realizar uma análise dos dados e verificar cada sinal do grupo de sinais suspeito. O procedimento para realizar a análise não é apresentado neste trabalho. Outro problema das abordagens de redes XOR que apresentam compactação de dados com perdas, é a necessidade de calcular os *golden vectors*. Contudo, muitos circuitos atuais possuem múltiplos domínios de *clock*, o que torna esse método pouco eficiente nesses casos.

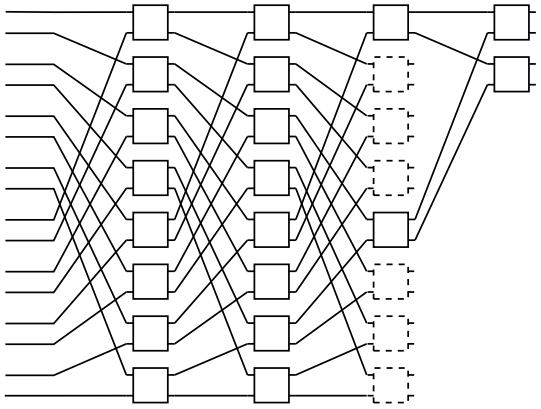
Em ambientes com múltiplos domínios de *clock* ou com caminhos multi ciclos, para criar *golden vectors* a análise deve ser tolerante a valores desconhecidos de registradores (bits-X). Liu e Xu (2012a) propõem uma nova rede mista capaz de lidar com *golden vectors* contendo bits-X nas assinaturas. A rede utiliza uma célula *XOR-MUX* para capturar erros. Na primeira vez que a depuração executa, a rede é configurada para XOR. Quando um erro é localizado, a rede pode ser reconfigurada (XOR-MUX) e passa a monitorar um estágio anterior. O processo itera pelos estágios da rede até encontrar o erro. Porém, utilizando essa abordagem em um ambiente indeterminístico, dependendo da quantidade de sinais explorados e da quantidade de sinais monitorados, o processo de depuração poderia levar muito tempo para completar. Em uma rede de tamanho 4096x32 são necessários pelo menos 7 iterações.

3.2 Redes Ômega assimétricas

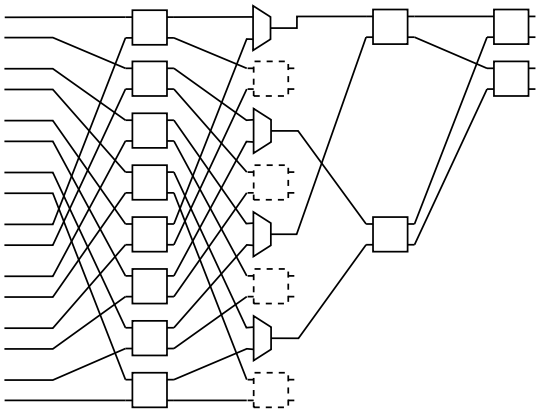
A rede proposta neste trabalho é gerada a partir de uma Ômega simétrica convencional. Para gerar essa rede, o trabalho propõe um método que remove os multiplexadores e *switches* não utilizados, começando pelo último estágio e que se propaga para os demais estágios. Esse processo é demonstrado na Figura 21 e detalhado no Algoritmo 1. Na Figura 21 é apresentado um exemplo do processo proposto para transformar uma rede Ômega 16x16 em uma rede Ômega assimétrica de tamanho 16x4. A rede resultante é composta por $N = 16$ sinais explorados e $M = 4$ sinais que são conectados e monitorados na memória intermediária.



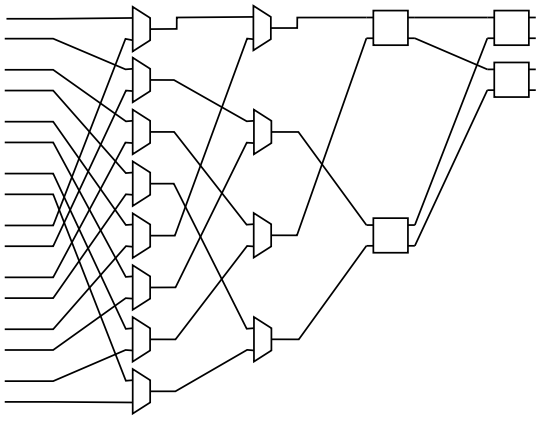
(a) Remoção dos multiplexadores do último estágio



(b) Remoção dos multiplexadores do terceiro estágio



(c) Remoção dos multiplexadores do segundo estágio



(d) Remoção dos multiplexadores do primeiro estágio

Figura 21 – Processo para gerar uma rede Ômega assimétrica. Os multiplexadores não utilizados são removidos, do último estágio para o primeiro. Quando um *switch* completo é removido, é possível propagar a remoção para um estágio anterior. A geração de uma rede Ômega assimétrica de 16x4 remove 44 multiplexadores.

A Figura 21a é o início do processo, uma rede Ômega simétrica de tamanho 16x16. Essa rede é originalmente composta por 32 *switches*. Na rede Ômega assimétrica, 12 saídas não serão utilizadas, logo, os respectivos *switches* não utilizados são removidos. Nesse exemplo, os 4 sinais no topo da rede são mantidos, assim, os 6 últimos *switches* são retirados. O método consiste em propagar para trás as remoções dos *switches* para os outros estágios. Portanto, o próximo passo consiste em remover os outros 6 *switches* que estão conectados aos *switches* que foram removidos (Figura 21b). Na Figura 21c, outros 4 *switches* são removidos, e 4 *switches* são trocados por multiplexadores 2x1. O último passo, que é apresentado na Figura 21d, é trocar 8 *switches* por 8 multiplexadores 2x1, resultando na rede de interconexão final. Considerando que 1 *switch* é composto por 2 multiplexadores 2x1, a rede Ômega assimétrica proposta é construída utilizando 20 multiplexadores.

Algoritmo 1 Remove *muxes* não utilizados na rede Ômega assimétrica

```
1:  mmx ← matriz de muxes inicializada com 1
2:  pttrn ← padraoinverso de conexao
3:  N ← numero de entradas
4:  outputs ← lista de saidas disponiveis
5:  ls ← ultimo estagio
6:
7:  for i ← 0 to N − 1 do
8:      mmx[ls][i] ← 0
9:  end for
10:
11: for i ← 0 to SIZEOF(outputs) − 1 do
12:     out ← outputs[i]
13:     mmx[ls][out] ← 1
14: end for
15:
16: for i ← ls to 1 do
17:     for j ← 0 to N/2 do
18:         if mmx[i][2 ∗ j] == 0 then
19:             if mmx[i][2 ∗ j + 1] == 0 then
20:                 mmx[i − 1][pttrn[2 ∗ j]] = 0
21:                 mmx[i − 1][pttrn[2 ∗ j + 1]] = 0
22:             end if
23:         end if
24:     end for
25: end for
```

No Algoritmo 1 é apresentado o método de propagação da remoção dos *switches*. A matriz **mmx** é utilizada para marcar se um determinado multiplexador está disponível. Antes do processo iniciar, todos os multiplexadores estão disponíveis, assim, na linha 1, os elementos da matriz são inicializados com **1**. Durante o processo, um vetor com o padrão de conexão *Shuffle Exchange* inverso é utilizado para verificar quais *switches* estão conectados a um *switch* removido (linha 2). **N** é a variável usada para guardar o número de entradas da rede. A variável **outputs** guarda a lista de saídas da rede que estão disponíveis. O laço de repetição na linha 7 remove todas as saída da rede e, na linha 11, outro laço utiliza a variável **outputs** para tornar as saídas da rede que foram especificadas disponíveis. Nas linhas 16–19, é realizado a iteração do último estágio para o primeiro, verificando se um *switch* 2x2 completo foi removido. Para verificar se um *switch* completo foi removido, basta testar se as duas saídas do *switch* (indexado por **j**) são iguais a **0**. Por fim, uma saída de cada *switch* que é conectado ao *switch* corrente é removido utilizando o vetor **pttrn** com o padrão inverso de conexão (linhas 20–21).

Nesse trabalho é assumido que na depuração pós-silício não é importante conectar um determinado sinal em uma porta específica na saída da rede, uma vez que os dados gravados na memória do *trace buffer* serão extraídos para analise e podem ser reordenados. Portanto, a taxa de bloqueio pode ser medida considerando que é possível conectar um determinado sinal a qualquer porta de saída disponível. O número de configurações possíveis para rotas em uma rede Ômega neste contexto é dado por $\binom{in}{out}$. Para exemplificar, existem $\approx 10^{80}$ possíveis combinações para uma rede Ômega de 4096x32. Portanto, nesse trabalho a taxa média de bloqueio é calculada de forma empírica utilizando uma heurística. Essa heurística é apresentada no Algoritmo 2.

Duas filas são utilizadas: **in_q** e **out_q**. A primeira guarda os sinais de entrada que devem ser roteados, e a segunda guarda as saídas disponíveis da rede. As variáveis **routed** e **failed** guardam, respectivamente, o número de sinais roteados e os não roteados. A variável **miss** é utilizada para contar o número de vezes que diferentes saídas da rede foram levadas em consideração no roteamento antes de estabelecer uma conexão ou determinar que um bloqueio ocorreu.

Na linha 9, um laço **enquanto** é utilizado para iterar até que a soma do número de rotas e o número de falhas seja igual à quantidade de saída da rede. Na linha 10, a função **ROUTE_PATH** tenta realizar o roteamento entre os primeiros sinais das filas de entrada e saída. Se os sinais foram roteados com sucesso (linhas 12–16), então, esses sinais são removidos das filas, o contador de sinais roteados é incrementado e a variável **miss** é restaurada para o valor inicial. Do contrário (linhas 17–26), se o roteamento falha, a porta de saída é realocada para o final da fila (linhas 18–19) e com a mesma porta de entrada é realizado uma nova tentativa de roteamento com a próxima saída disponível da fila. Esse processo se repete até todas as saídas serem testadas (linha 21). Se uma conexão

Algoritmo 2 Determinar a taxa de bloqueio nas redes Ômega assimétricas

```
1:  function omega_blkng_rate
2:      in_q ← fila dos sinais de entrada
3:      out_q ← fila das saidas disponiveis
4:      routed ← 0
5:      failed ← 0
6:      miss ← 0
7:      blkng_rate ← 0
8:
9:      while routed + failed < numero de saidas do
10:         r ← ROUTE_PATH(FRONT(in_q), FRONT(out_q))
11:
12:         if r == TRUE then
13:             POP_FRONT(in_q)
14:             POP_FRONT(out_q)
15:             routed ← routed + 1
16:             miss ← 0
17:         else
18:             PUSH_BACK(FRONT(out_q))
19:             POP_FRONT(out_q)
20:             miss ← miss + 1
21:             if miss ≥ SIZE(out_q) then
22:                 POP_FRONT(in_q)
23:                 failed ← failed + 1
24:                 miss ← 0
25:             end if
26:         end if
27:
28:     end while
29:
30:     blkng_rate ← routed/numero de saidas
31:     return blkng_rate
32: end function
```

não pode ser estabelecida para nenhuma saída, um bloqueio é determinado (linha 23). Na linha 30, a taxa de bloqueio é calculada. Para alcançar resultados mais consistentes, esse algoritmo pode ser executado diversas vezes com diferentes conjuntos de entrada/saída a fim de calcular uma taxa média de bloqueio.

Nas redes Mux Trees existe apenas uma única saída disponível para várias en-

tradas. Portanto a taxa de bloqueio é facilmente determinada roteando os determinados sinais do conjunto de entrada. Esse processo é apresentado no Algoritmo 3.

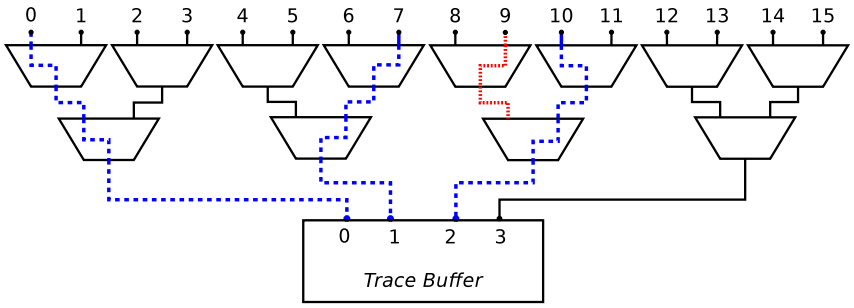
Algoritmo 3 Determinar a taxa de bloqueio nas redes Mux Trees

```
1:  function mux_tree_blkng_rate
2:      in_q ← queue of inputs signals
3:      routed ← 0
4:      failed ← 0
5:      blkng_rate ← 0
6:
7:      while routed + failed < SIZEOF(in_q) do
8:          r ← ROUTE_PATH(FRONT(in_q))
9:
10:         if r == TRUE then
11:             routed ← routed + 1
12:         else
13:             failed ← failed + 1
14:         end if
15:         POP_FRONT(in_q)
16:
17:     end while
18:
19:     blkng_rate ← routed/number of outputs
20:     return blkng_rate
21: end function
```

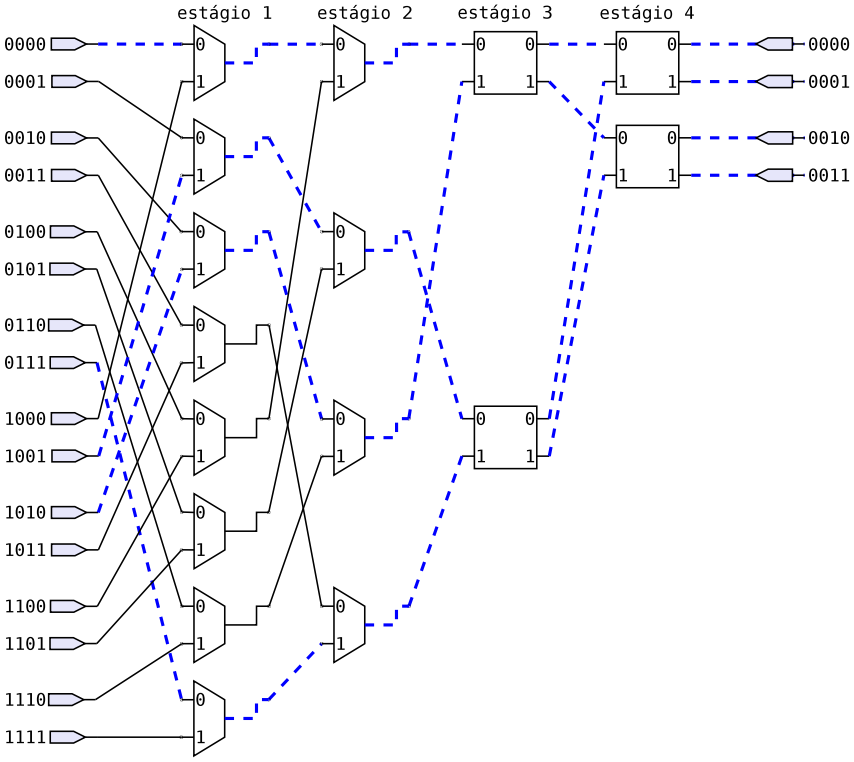
Uma fila, **in_q**, guarda os sinais de entrada selecionados. **Routed** e **failed** são as variáveis utilizadas para guardar o número de conexões roteadas e não roteadas respectivamente (linhas 3–4). Na linha 7, um laço de repetição **enquanto** itera até que o número de rotas estabelecidas somada ao número de rotas com falhas seja igual ao tamanho do conjunto de entrada desejado pelo projetista. Na linha 19 é calculada a taxa de bloqueio para a rede Mux Tree, que é o mesmo cálculo realizado para a rede Ômega assimétrica.

Nas Figuras 22a e 22b, são apresentadas as redes Mux Tree 16x4 e Ômega 16x4. No exemplo, o mesmo conjunto de entrada, composto por {0, 7, 9, 10}, apresenta diferentes taxas de bloqueio. Na rede Ômega, todos os sinais foram roteados, logo 0% de taxa de bloqueio. Na rede Mux Tree, houve 25% de bloqueio. Observe que se o conjunto de entrada fosse alterado para {0, 8, 9, 10}, haveria 50% de taxa de bloqueio na Mux Tree e 25% na Ômega.

Para essas instâncias das redes Ômega e Mux Tree apresentadas na Figura 22, a taxa média de bloqueio é $\approx$ 27% para ambas as redes. Para determinar a taxa média



(a) Rede Mux Tree 16x4. Para esse conjunto de entrada, houve 25% de taxa de bloqueio. O conjunto de entrada contém as portas 9 e 10 que passam pelo multiplexador no segundo estágio.



(b) Rede Ômega 16x4. Para esse conjunto de entrada, a taxa de bloqueio é 0%.

Figura 22 – Exemplo de roteamento para Ômega e Mux Tree 16x4.

de bloqueio, deve-se observar a variação da taxa de bloqueio em função do número de amostras. Uma amostra é um conjunto de sinais de entrada que é utilizado como parâmetro nos Algoritmos 2 e 3. Quando um grande número de amostras é aplicado nos algoritmos, a taxa de bloqueio converge para a média, e quando o número de amostras é grande, a variação da taxa é $\approx 0\%$. Esse procedimento pode ser observado na Figura 23. O gráfico apresenta no eixo y a taxa média de bloqueio, e no eixo x a quantidade de amostras. O eixo x está em escala log_{10} e apresenta um conjunto de 100 mil amostras. Observe que

existe uma grande variação na taxa média de bloqueio quando o conjunto de amostras é pequeno.

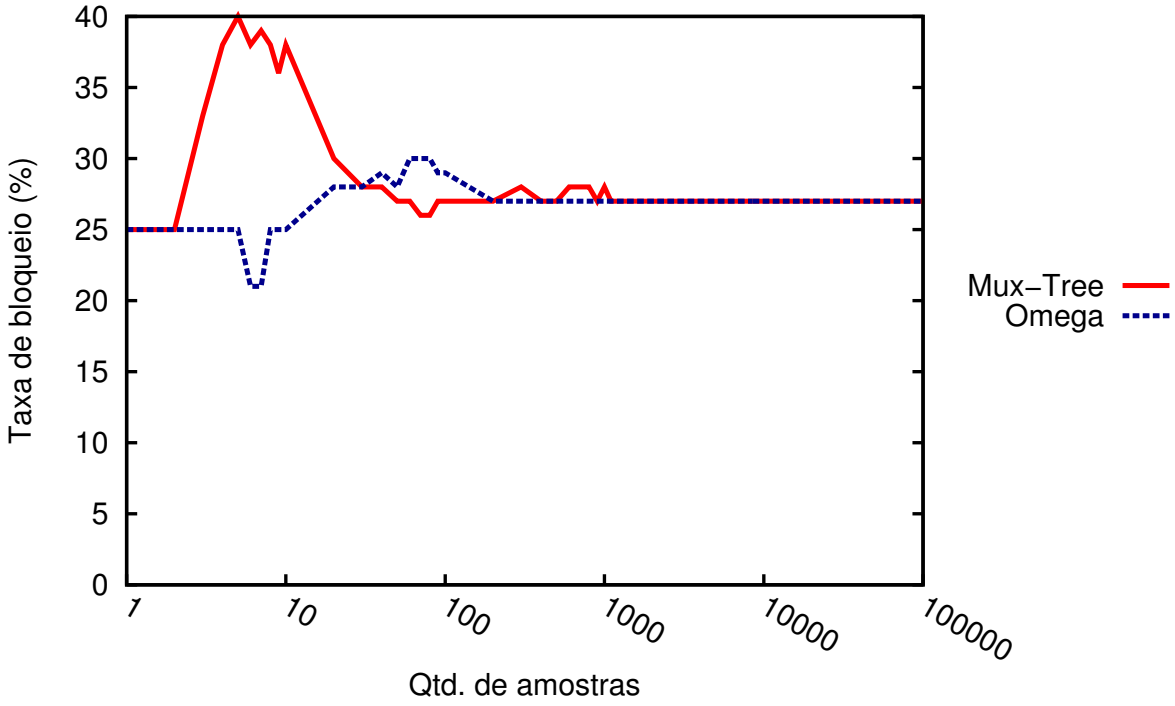
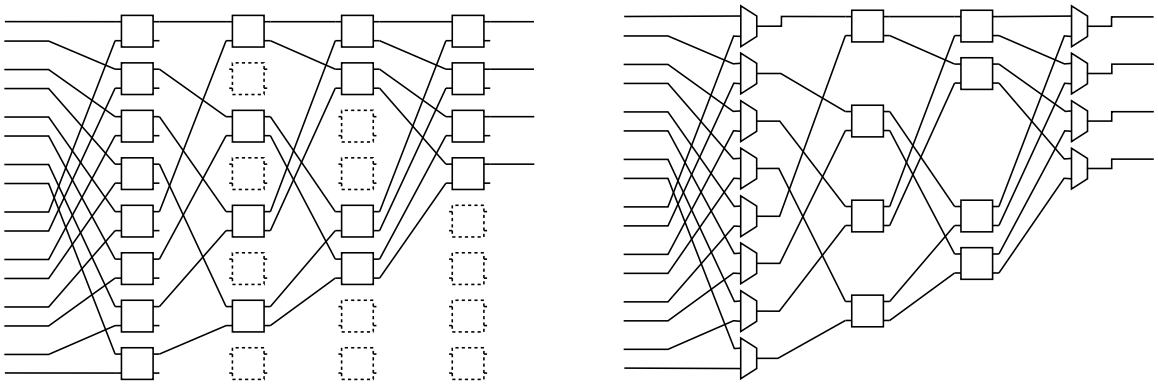


Figura 23 – Método para determinar a taxa média de bloqueio. O gráfico no eixo y apresenta a taxa média de bloqueio da rede, e no eixo x a quantidade de amostras utilizadas para determinar a taxa. O eixo x está em escala log_{10}, assim, deve-se observar que a maior parte das amostras está na direita, e que no último trecho existem 90 mil amostras.

Para reduzir a taxa média de bloqueio nas redes Ômega assimétricas, é possível dispersar as saídas da rede. No entanto, menos multiplexadores serão removidos, isto porque o processo de remoção multiplexadores precisa que um *switch* completo seja removido para propagar a remoção para outros estágios. O processo para remover os multiplexadores não utilizados apresentado no Algoritmo 1 funciona para qualquer configuração de saídas da rede.

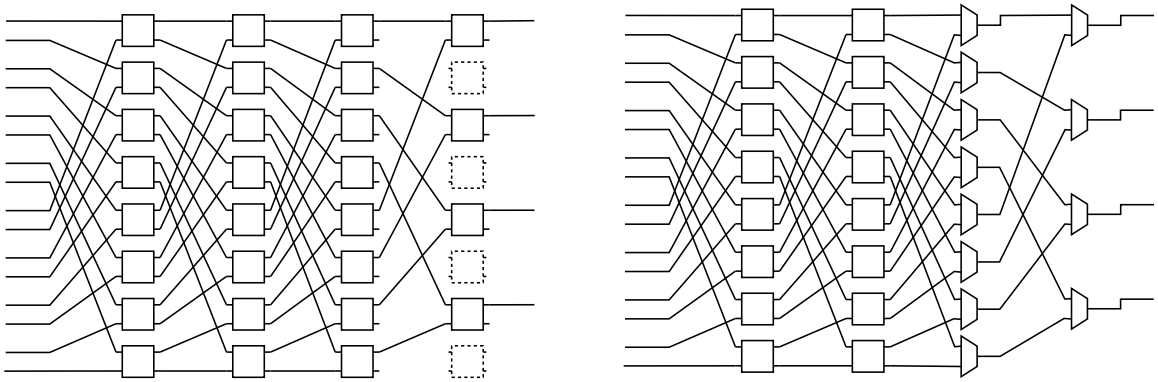
Nas Figuras 24 e 25 são apresentados dois possíveis arranjos das saídas da rede, com uma saída disponível a cada 2 e uma a cada 4. Na Figura 24, as saídas {0, 2, 4, 6} foram escolhidas, 28 multiplexadores 2x1 foram utilizados. Na Figura 24, as saídas {0, 4, 8, 12} foram escolhidas, aumentando o número de multiplexadores para 44. Para facilitar as comparações entre as diversas possibilidade de configuração da rede Ômega assimétrica, nesse trabalho as redes recebem o sufixo Gi, sendo i o número utilizado para indexar as saídas disponíveis da rede. Assim, a configuração da rede Ômega assimétrica com uma saída disponível a cada 2 é chamada de Ômega G2 (Figura 24), e a configuração a cada 4 é chamada de Ômega G4 (Figura 25).

O Algoritmo 2 não precisa de qualquer modificação para lidar com esses diferentes arranjos na saída da rede. Logo, utilizando o mesmo processo, a taxa média de bloqueio computada para o arranjo na Figura 24b é de $\approx 10\%$. Na Figura 25b, a taxa média de bloqueio é reduzida para $\approx 0\%$.



(a) (b)

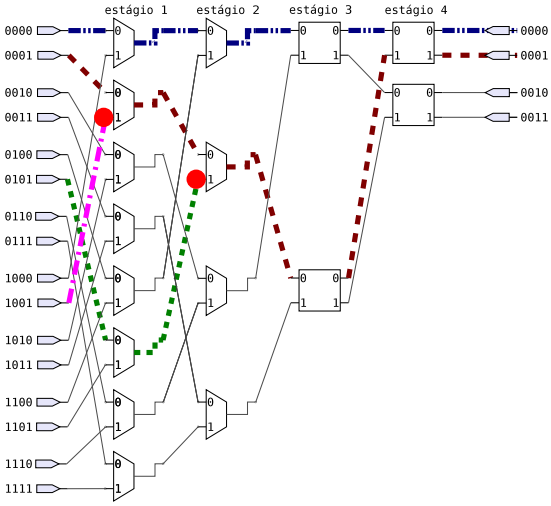
Figura 24 – Rede Omega 16x4 com 28 multiplexadores 2x1. A taxa média de bloqueio calculada para essa rede é $\approx 10\%$.



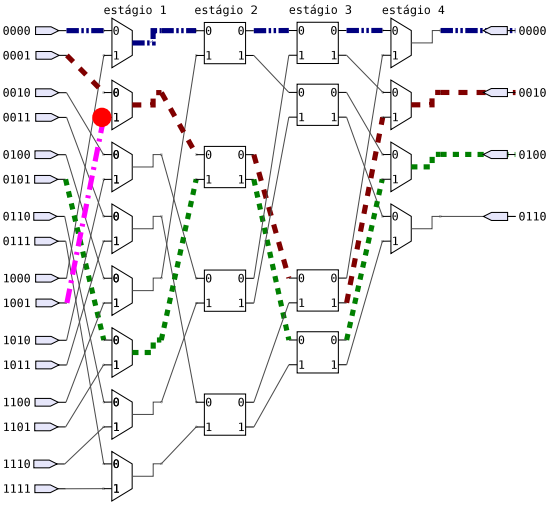
(a) (b)

Figura 25 – Rede Omega 16x4 com 44 multiplexadores 2x1. A taxa média de bloqueio calculada para essa rede é $\approx 0\%$.

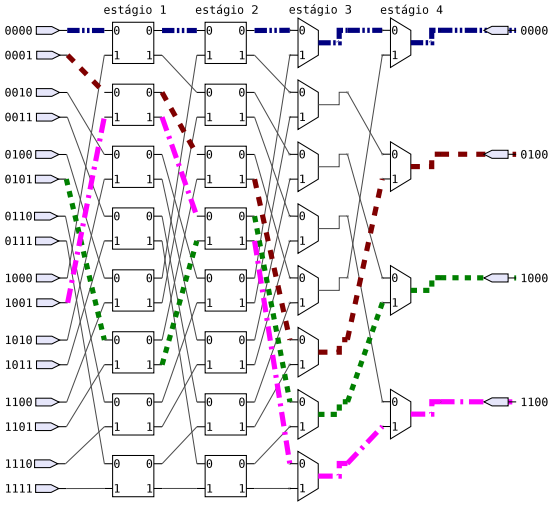
Para ilustrar como a dispersão dos multiplexadores impacta na taxa de bloqueio, observe as redes que são apresentadas na Figura 26. O conjunto de entrada para essas redes é composto por $\{0, 1, 5, 9\}$. A rede da Figura 26a é capaz de rotear $0 \rightarrow 0$ e $1 \rightarrow 1$, mas não é possível rotear $5 \rightarrow 2$ ou $5 \rightarrow 3$; também não é possível conectar $9 \rightarrow 2$ ou $9 \rightarrow 3$. Na rede apresentada em 26a, não é possível rotear a porta 5 pois ocorre um bloqueio com outra a rota $001_b \rightarrow 001_b$ no segundo estágio. A porta de entrada 9 também está bloqueada pela mesma rota já estabelecida logo no primeiro estágio. A rede da Figura 26b conecta $0 \rightarrow 0$, $1 \rightarrow 2$ e $5 \rightarrow 4$, mas não é possível rotear $9 \rightarrow 6$. Novamente, no mesmo ponto ocorre o bloqueio para a porta 9. A rede apresentada na Figura 26c é capaz de rotear $0 \rightarrow 0$, $1 \rightarrow 4$, $5 \rightarrow 8$ e $9 \rightarrow 12$. Nesses exemplos, a taxa de bloqueio para



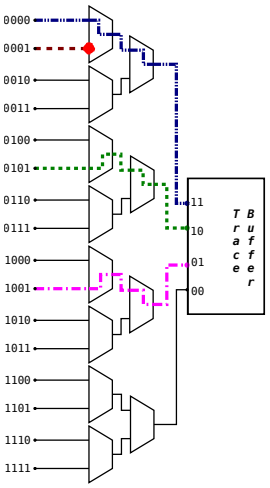
(a) Rede Ômega assimétrica com as saídas agrupadas no topo.



(b) Rede Ômega assimétrica com incremento de 2 na saída (Ômega G2).



(c) Rede Ômega assimétrica com incremento de 4 na saída (Ômega G4).



(d) Rede Mux Tree.

Figura 26 – Comparação de diferentes arranjos nas redes Ômega assimétricas. A dispersão das saídas da rede aumenta o número de multiplexadores disponíveis e reduz o bloqueio. Em a) o bloqueio é 50%, em b) 25% e em c) 0%. Em d) uma rede Mux Tree apresenta 25% de bloqueio para o mesmo conjunto de entrada.

cada uma das redes é respectivamente 50%, 25% e 0%. A taxa de bloqueio para uma rede Mux Tree 16x4 com o mesmo conjunto de entrada é 25%, conforme é apresentado na Figura 26d.

3.3 Geração automática das redes de interconexão

Nesta seção são apresentados os métodos para a geração das redes de interconexão Crossbar, Ômega, Clos, Benes e Mux Tree. Os métodos propostos nessa seção foram implementados na ferramenta **Vericonn** (***Veri****log* *Inter****conn****ection*) que gera automaticamente as redes de interconexão em LDH Verilog. A utilização de uma ferramenta para geração automática de redes de interconexão pode ajudar a reduzir os custos na fase de depuração do dispositivo, ampliando a receita e reduzindo o tempo de entrega do produto para o mercado.

Os dispositivos de interconexão gerados pela ferramenta fornecem uma interface simples para qualquer rede gerada. Redes RIMs são construídas pela conexão de diversas redes Crossbar. Para configurar as redes RIMs, basta configurar cada Crossbar interno. Dessa forma, a interface dos módulos é simples. Comparado com a interface do Crossbar, as redes RIMs possuem apenas um barramento adicional para especificar qual Crossbar está sendo configurado.

Nesse trabalho é considerado que toda a configuração de roteamento, incluindo a configuração interna dos Crossbares, é gerado por um software externo. O tempo gasto para configuração de uma RIM depende das própria característica da rede. Uma simples Crossbar requer N ciclos de *clock*, N sendo o número de saídas do Crossbar. Uma rede Mux Trees cujo tamanho é potência de 2 requer $\sum_{i=1}^{log_2(in)-log_2(out)} in/(2^i)$ ciclos de *clock* para configurar. Para exemplificar, uma rede Crossbar de tamanho 1024x32 gasta 32 ciclos de *clock* para configurar, e uma rede Mux Trees de 1024x32 gasta 992 ciclos de *clock* para configurar.

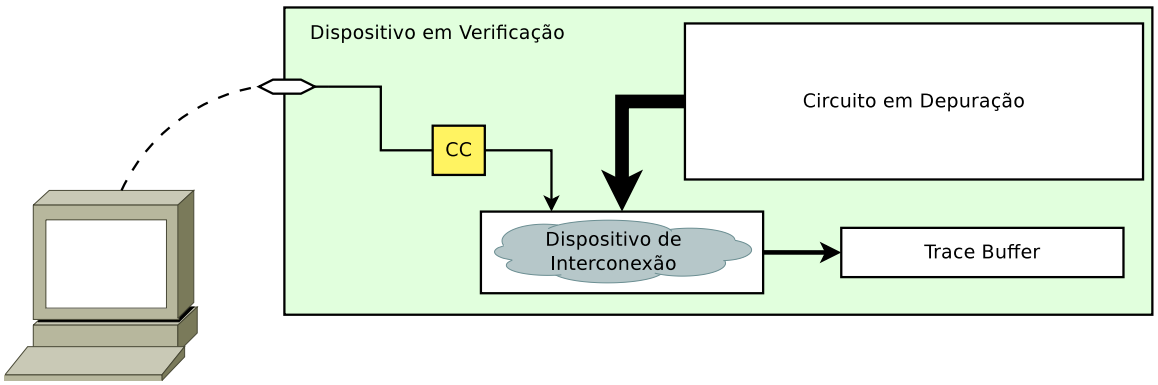


Figura 27 – Módulo de configuração da rede de interconexão. A configuração da rede é gerada por um software externo. O módulo Carregador de Configuração (CC) faz a desserialização dos dados de configuração da rede de interconexão.

O dispositivo de interconexão gerado deve ser configurado antes do processo de depuração pós-silício iniciar. Na Figura 27, um software externo é utilizado para gerar a configuração da rede de interconexão. O computador envia os dados de configuração para

o Dispositivo em Verificação (DeV) e é recebido pelo módulo Carregador de Configuração (CC). O módulo CC configura o **Dispositivo de Interconexão**. Quando o processo de depuração iniciar, os sinais observados passam pelo Dispositivo de Interconexão e são armazenadas na memória intermediária de rastreio (*Trace Buffer*). Supondo o caso mais simples, que a configuração da rede é enviada por uma conexão serial, o módulo CC é necessário para desserialização do processo.

3.3.1 A arquitetura para a rede Crossbar

O módulo Crossbar gerado pelo Vericonn é uma instância da arquitetura apresentada pela Altera Corporation (2004). O módulo contém três características. O módulo **Matriz de Switches** recebe os dados de entrada, e a saída é dada de acordo com a configuração dos registradores. O módulo **Decodificador** decodifica o endereço de saída. O módulo **Configuração de Registradores** controla a reconfiguração dos registradores de saída. Esta arquitetura é apresentada na Figura 28. Para configurar a rede Crossbar, deve-se colocar os dados de **endereço de entrada** e **endereço de saída** no barramento enquanto o sinal **configura** está alto. Na Figura 28, o círculo apresenta as conexões da matriz de switches em detalhes. O barramento de entrada é conectado em todos os switches. Cada switch é configurado por registradores de configuração. A saída do switch é um fio. O módulo matriz de switches une cada saída dos switches internos para formar o barramento de saída.

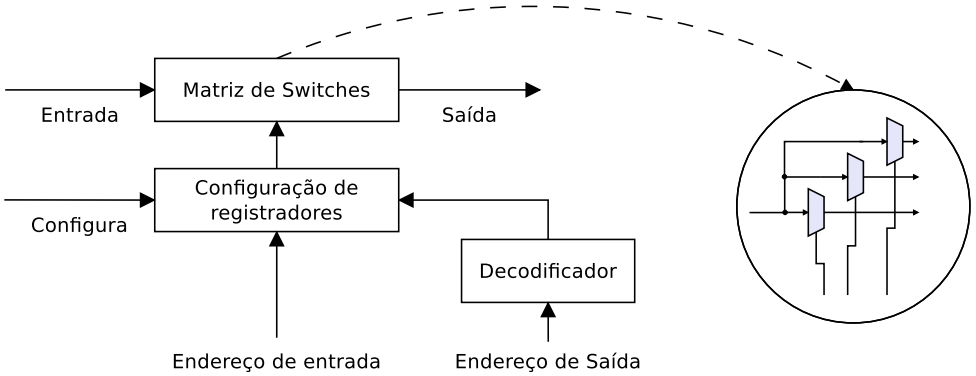


Figura 28 – Arquitetura da rede Crossbar reconfigurável. A saída da rede é controlada pelos registradores de configuração. O sinal **configura** ativa o modo de configuração e o módulo **Configuração de Registradores** configura a **matriz de switches** utilizando os **endereços de entrada** e **saída**.

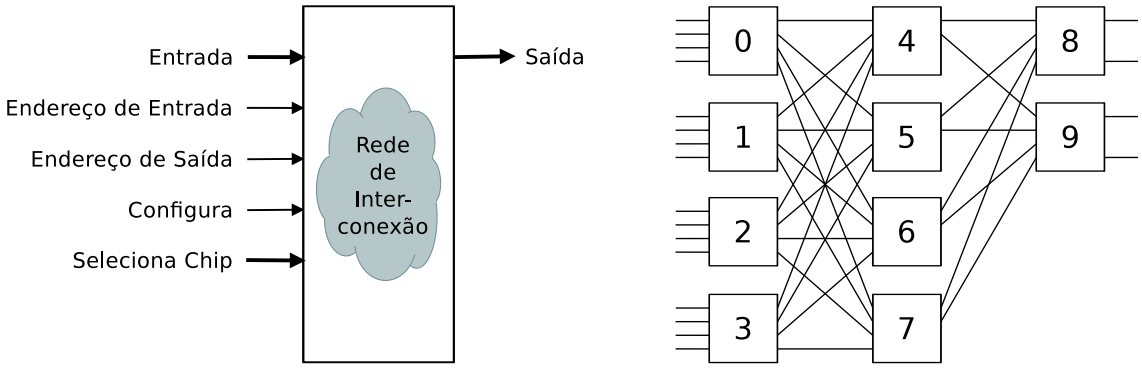
A arquitetura da Crossbar original (Altera Corporation, 2004) utiliza um esquema de configuração em duas fases, nomeado *load/config*. Primeiro, a configuração é carregada para os registradores, e no segundo passo a configuração é aplicada. O Vericonn gera a rede Crossbar tanto com o esquema *load/config* como o esquema de configuração direta.

Vale ressaltar que o custo de área do esquema *load/config* é mais alto porque todos os registradores de configuração são duplicados.

O Apêndice A apresenta um módulo Crossbar 4x4 gerado pela ferramenta. Cinco arquivos e módulos são gerados para refletir a arquitetura apresentada nessa seção. O módulo **Decoder4** é o módulo decodificador do endereço de saída para configuração da rede. O módulo **Switch4** é utilizado para realizar a configuração dos registradores. **SwitchMatrix4x4** interconecta vários módulos **IOCell4**, como uma matriz de multiplexadores. O módulo topo é o **DSM4x4** que interliga todos os módulos. Nesse apêndice também é apresentada uma simulação que demonstra como configurar o módulo DSM4x4.

3.3.2 A arquitetura para redes multiestágio (RIM)

Todas as RIMs implementadas utilizam os mesmos sinais do módulo Crossbar e mais o barramento **Seleciona Chip**. Esse barramento é utilizado para selecionar qual o Crossbar está em configuração. Somente um Crossbar pode ser configurado em um determinado tempo. A Figura 29a apresenta a arquitetura do módulo. Todos os sinais de entrada de configuração passam através da interface da RIMs diretamente para cada Crossbar. O barramento de entrada de dados é distribuído para as entradas dos Crossbares do primeiro estágio. As saídas de dados dos Crossbares do último estágio é concatenada para formar o dado de saída. Se for desejado, a ferramenta pode gerar um módulo para traduzir o endereço no barramento **Seleciona Chip**. Sem a tradução de endereços, o barramento possui um sinal para cada Crossbar interno.



(a) Implementação da RIMs.

(b) Endereçamento no barramento Seleciona Chip.

Figura 29 – Arquitetura das redes multiestágio gerados pelo Vericonn. Em (a), a arquitetura das redes multiestágio gerados pelo Vericonn apresenta a mesma interface da Crossbar mais o barramento Seleciona Chip. Em (b), a forma de endereçamento das redes Crossbares interna: por estágio e bit menos significativo.

Na Figura 29b é apresentado o endereçamento interno das Crossbares para uma rede RIM. O primeiro Crossbar é aquele que está conectado no bit menos significativo do

barramento de entrada de dados. O último Crossbar do primeiro estágio está conectado no bit mais significativo do barramento de dados. Assim, o endereçamento é sequencial e por estágio. Se comparado à uma matriz, o endereçamento é primeiro por coluna. Observe que o endereçamento dos *switches* começa de 0. Quando não há tradução de endereços no barramento **Seleciona Chip**, se bit menos significativo do barramento está ativo, a configuração no *switch* 0 está ativa.

3.3.2.1 Redes Clos

Mesmo quando a capacidade de permutação é desejada, não há necessidade de se utilizar uma rede Clos não bloqueante no contexto de depuração pós-silício. Como o roteamento é realizado antes do processo de depuração começar, uma rede Clos rearranjável é adequada para depuração pós-silício. Na ferramenta apresentada, é possível gerar tanto redes Clos não bloqueantes como redes Clos rearranjável.

Cada Crossbar do primeiro estágio tem tamanho $n_1 = N^{2/(s+1)}$, n_1 sendo o número de sinais de entrada da Crossbar, **N** o número de entradas, e **s** o número de estágios. Similarmente, $n_2 = M^{2/(s+1)}$, n_2 sendo o número de saídas da Crossbar e **M** o número de saídas das rede. Para casos onde $N \equiv r(mod\ n)$, é gerado um Crossbar extra para o resto **r**, para conectar todas as entradas e saídas no primeiro e terceiro estágio, respectivamente.

O Apêndice B apresenta uma simulação para configurar um módulo Clos 16x4 gerado pelo Vericonn. No apêndice estão presentes os códigos em Verilog do módulo topo e da simulação em LDH Verilog.

3.3.2.2 Redes Mux Tree

Para gerar redes Mux Trees de tamanhos arbitrários, a ferramenta proposta precisa lidar com casos onde ocorrem números ímpares de sinais para se conectar a um determinado estágio da rede. Esses números ímpares de sinais podem ocorrer tanto na entrada da rede, quanto em um estágio intermediário. O primeiro caso ocorre quando o projetista está selecionando o conjunto de sinais a ser observado e apresenta um número resultante ímpar de sinais. Para exemplificar o segundo caso, considere uma rede Mux Tree 500x36. No primeiro estágio, 250 sinais são guiados ao segundo estágio. Do segundo estágio, 125 sinais são guiados para o terceiro estágio. Na abordagem, apresentada nesse trabalho, o terceiro estágio é composto por 61 multiplexadores 2x1. O último multiplexador é instanciado com tamanho 3x1. Na Figura 30 é apresentado uma rede Mux Tree 11x2. Nesse exemplo, números ímpares ocorrem nos dois estágios da rede.

O segundo caso que o método precisa lidar é com tamanho de rede que proporcionam estágios incompletos. Para entender o que é chamado de estágio incompleto, considere uma rede Mux Tree de 1000x100. No primeiro estágio, 500 sinais são guiados para o próximo estágio; 250 sinais vão para o terceiro estágio, e 125 para o quarto estágio. Se todos

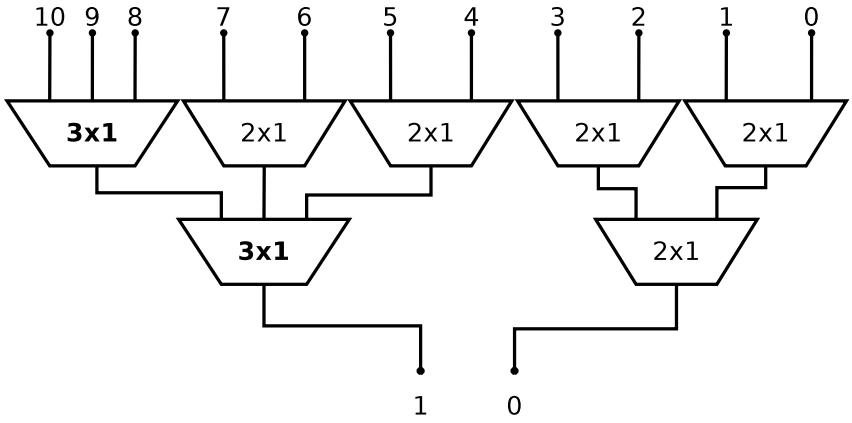


Figura 30 – Exemplo de rede Mux Tree 11x2. Observe que para lidar com um número ímpar na entrada da rede, é instanciado um multiplexador de tamanho 3x1. Na entrada do segundo estágio também ocorre um número ímpar de sinais, e novamente é instanciado um multiplexador de tamanho 3x1.

os sinais passassem por multiplexadores, na saída da rede haveria um número de sinais inferior ao número desejado. Logo, no último estágio apenas 50 sinais precisariam passar por multiplexadores. Os sinais restantes podem apenas passar direto do quarto estágio para a saída da rede. Na Figura 31 é apresentado uma rede Mux Tree de 16x6. Neste exemplo, somente 2 multiplexadores são necessários no segundo estágio.

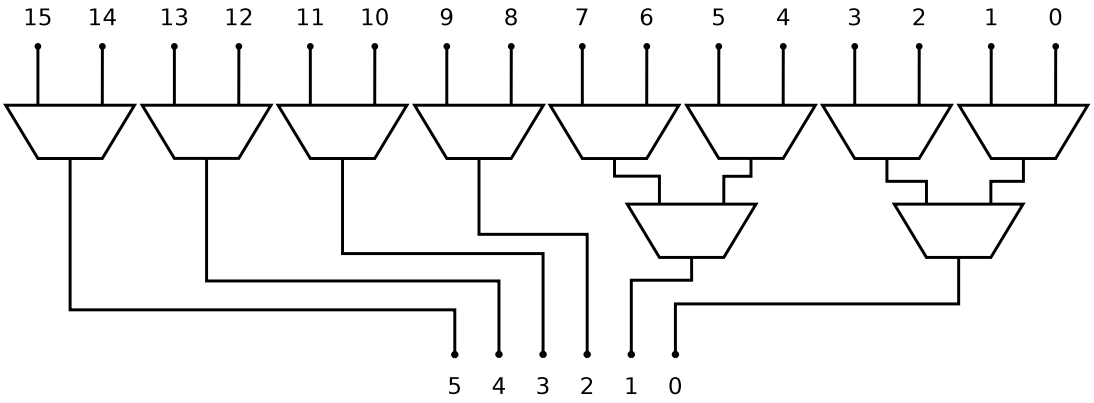


Figura 31 – Exemplo de rede Mux Tree 16x6. No segundo estágio, nem todos os sinais provindos do antecessor devem passar por multiplexadores. Nesse exemplo, apenas dois multiplexadores são necessários no segundo estágio.

No Algoritmo 4 é apresentado o processo para gerar as redes Mux Trees de tamanhos arbitrários. A variável **restante** armazena o número de sinais restantes que devem ser conectados a um estágio ou à saída da rede. Na linha 2 **restante** é iniciado com o total de entradas da rede. A variável **estagio** armazena o atual estágio sendo aprocessado e é inicializado com 0 na linha 3. Nas linhas 4–18, o laço itera até que não haja possibilidade de gerar outro estágio em que todos os sinais são guiados para multiplexadores. Na Figura 31, somente o primeiro estágio é gerado pelo laço de repetição. Para cada iteração,

Algoritmo 4 Instancia redes Mux Trees de tamanhos arbitrários

```
1: tamanho_saida ← tamanho da saida da rede
2: restante ← tamanho da entrada da rede
3: estagio ← 0
4: while restante ≥ tamanho_saida * 2 do
5:     n_muxes ← restante ÷ 2
6:     if restante % 2 == 0 then
7:         for i = 1 to n_muxes do
8:             DECLARE_MUX_2X1(estagio, i)
9:         end for
10:    else
11:        for i = 1 to n_muxes − 1 do
12:            DECLARE_MUX_2X1(estagio, i)
13:        end for
14:        DECLARE_MUX_3X1(estagio, n_muxes)
15:    end if
16:    restante ← n_muxes
17:    estagio ← estagio + 1
18: end while
19: if restante > tamanho_saida then
20:    n_muxes ← restante − tamanho_saida
21:    for i = 1 to n_muxes do
22:        DECLARE_MUX_2X1(estagio, i)
23:        s ← OUT_SIGNAL(estagio, i)
24:        TO_OUT_PORT(s, out_port[i])
25:    end for
26:    r ← n_muxes * 2
27:    for i = n_muxes to tamanho_saida do
28:        s ← OUT_SIGNAL(estagio − 1, r)
29:        TO_OUT_PORT(s, out_port[i])
30:        r ← r + 1
31:    end for
32: else
33:    for i = 1 to tamanho_saida do
34:        s ← OUT_SIGNAL(estagio, i)
35:        TO_OUT_PORT(s, out_port[i])
36:    end for
37: end if
```

nas linhas 6–15, o algoritmo verifica se o estágio é composto por somente multiplexadores 2x1, ou se contém um multiplexador 3x1 no caso de um número ímpar de sinais. Observe que na linha 5 a variável *n_muxes* é inteira. Nas linhas 16–17 as variáveis **restante** e **estagio** são atualizadas para decrementar a quantidade restante de multiplexadores a serem instanciados e incrementar o valor do estágio para processamento.

Depois do laço de repetição, na linha 19 é verificado se a rede está completa: **restante** = **tamanho_saida**. Quando a rede está completa após o laço de repetição, é necessário apenas conectar a saída do último estágio na saída da rede. Esse processo ocorre nas linhas 32–37. Quando a rede não está completa após o laço (linhas 19–31), o último estágio não terá todos os sinais do estágio anterior guiados para multiplexadores. O número de sinais que devem ser guiados para multiplexadores é armazenado em **n_muxes**, linha 20. Em seguida, nas linhas 21–25, todos os multiplexadores requeridos são declarados e as respectivas saídas são conectadas às portas de saída da rede. Os sinais restantes, que não são guiados por multiplexadores no último estágio, são recuperados do estágio anterior (**estagio - 1**). O índice inicial para conectar esses sinais nas portas de saída é definido em **r**, linha 26. Nas linhas 27–31 os sinais que não passam por multiplexadores no último estágio são conectados a porta de saída da rede.

O Apêndice C apresenta uma simulação para configurar um módulo Mux Tree 16x4 gerado pelo Vericonn. No apêndice estão presentes os códigos em Verilog do módulo topo e da simulação em LDH Verilog.

3.3.2.3 Redes Ômega

Para gerar redes Ômega de tamanhos arbitrários, a ferramenta utiliza o Algoritmo 5. O algoritmo utiliza os conceitos apresentados na seção 3.2. Na linha 1, a variável **mmx** é inicializada com a matriz de multiplexadores disponíveis na rede. Essa matriz é determinada pelo Algoritmo 1 apresentado na seção 3.2. Na linha 2, **pdr_conn** é inicializado com uma estrutura de mapa, cuja chave é o índice do multiplexador e o valor representa o índice da conexão proveniente de outro estágio. A variável **estagio**, na linha 3 é o índice do estágio sendo processado no algoritmo. A variável **tam_entrada** é inicializada com o tamanho da entrada da rede. E o tamanho de cada Crossbar da rede é guardado na variável **radix**, linha 5.

Nas linhas 7–10, um laço itera para declarar os fios que são conectados às portas de entrada da rede Ômega. No laço seguinte, linhas 12–34, a rede é instanciada estágio por estágio. O laço nas linhas 13–33 itera para gerar cada Crossbar de um determinado estágio da rede Ômega. Na linha 14, uma lista com as saída disponíveis do Crossbar é inicializada com vazio. Nesse trabalho analisamos apenas redes Ômega com radix de tamanho 2, porém outros tamanhos são possíveis e o algoritmo apresentado é capaz de gerar redes com outros tamanhos de radix. Considerando uma rede com radix 2, a variável

Algoritmo 5 Instancia redes Ômega de tamanhos arbitrários

```
 1: mmx ← matriz de muxes
 2: pdr_con ← mapa com o padrao de conexao da omega
 3: estagio ← 0
 4: tam_entrada ← tamanho da entrada da rede
 5: radix ← tamanho do crossbar
 6:
 7: for i = 0 : tam_entrada do
 8:     fio ← DECLARA_FIO(i, estagio)
 9:     CONECTA_PORTA(fio, i)
10: end for
11:
12: for estagio = 1 : quantidade de estagios da rede do
13:     for crossbar = 0 : tam_entrada ÷ radix do
14:         lista_saida ← {}
15:         for r_i = 0 : radix do
16:             m_i = radix * crossbar + r_i
17:             mux = mmx[estagio − 1][m_i]
18:             if mux == MUX_FREE then
19:                 fio ← DECLARA_FIO(mux, estagio)
20:                 ADICIONA_LISTA(lista_saida, fio)
21:             end if
22:         end for
23:         if TAMANHO(lista_saida) > 0 then
24:             lista_entrada ← {}
25:             for r_i = 0 : radix do
26:                 m_i = radix * crossbar + r_i
27:                 mux = pdr_con[m_i]
28:                 fio ← FIO_ENTRADA(mux, estagio)
29:                 ADICIONA_LISTA(lista_entrada, fio)
30:             end for
31:             DECLARA_CROSSBAR(estagio, crossbar, lista_entrada, lista_saida)
32:         end if
33:     end for
34: end for
```

lista_saida tem um ou dois fios, dependendo se naquela posição há um *switch* 2x2 ou um *switch* 2x1. A lista pode ser vazia se o Crossbar tiver sido removido. Nas linhas 15–22, um laço itera verificando cada multiplexador do Crossbar. Na linha 16, é determinado

o índice do multiplexador sendo verificado. Na linha 17, o algoritmo busca o valor do multiplexador na matriz de multiplexadores, que pode ser 0 ou 1 (não existe ou existe). Nas linhas 18–21, caso o multiplexador exista, um fio é declarado (linha 19) e em seguida adicionado na lista de saída disponível (linha 20).

Nas linhas 23–32, caso o Crossbar sendo processado exista, a lista com as saídas do Crossbar não será vazia, então o bloco é processado. Uma segunda lista é utilizada para armazenar os fios de entrada do Crossbar, na linha 24, a variável **lista_entrada** é declarada para esse fim. O laço nas linhas 25–30, busca quais serão os fios conectados à rede Crossbar sendo processada. Na linha 26, é determinado o índice do multiplexador sendo processado. Na linha 27 o algoritmo busca no mapa qual é o multiplexador que conecta na entrada do índice sendo processado. Em seguida, é determinado o nome do fio desse multiplexador (linha 28). Na linha 29, o fio é adicionado na lista de fios de entrada. Por fim, na linha 31, uma Crossbar é declara para um determinado estágio.

O Apêndice D apresenta uma simulação para configurar um módulo Ômega 16x4 gerado pelo Vericonn. No apêndice estão presentes os códigos em Verilog do módulo topo e da simulação em LDH Verilog.

4 Resultados

Neste capítulo, os resultados obtidos são apresentados em três seções. Na seção 4.1 é apresentado um comparativo de área e taxa de bloqueio entre as redes Ômega assimétricas proposta, redes Mux Trees e redes Clos. Na seção 4.2 são apresentados os custos extras para inclusão das redes de interconexão dentro de projetos reais.

4.1 Comparação de custos de área e taxa de bloqueio

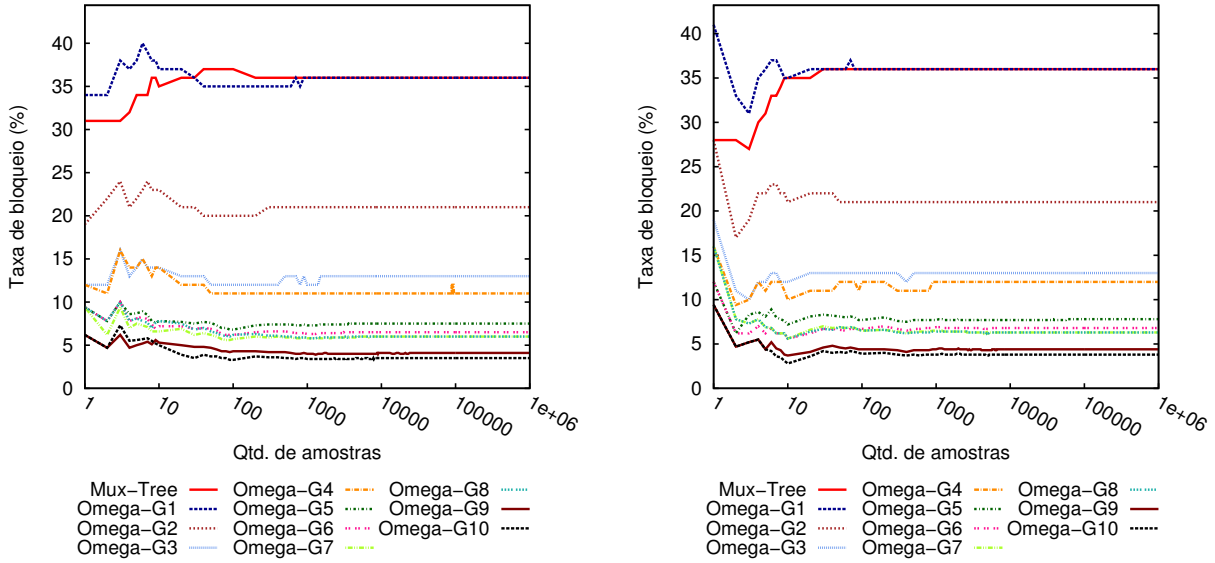
Nos experimentos realizados foram comparados a taxa média de bloqueio e área das redes Mux Tree, Ômega e Clos de vários tamanhos. Para cada topologia de rede, os códigos em LDH Verilog foram gerados pela ferramenta proposta **Vericonn** e então esses códigos foram sintetizados utilizando a ferramenta **Encounter RTL Compiler 11.25** (Cadence Design Systems, 2015).

A área apresentada é dada em termos de quantidade de portas NANDs equivalentes. Os valores apresentados nesse trabalho levam em consideração a área total do circuito com portas e fios. Esse valor é calculado pela ferramenta de síntese. Para obter o valor em NANDs equivalentes, a área total é dividida pelo tamanho da maior célula NAND da biblioteca de células utilizada na síntese. Dessa forma, foi utilizado a célula NAND2 cujo tamanho é $8.96\mu m * 6.72\mu m = 60.211\mu m$ (IBM Systems and Technology Group, 2010, p. 48).

Em cada gráfico apresentado nesse capítulo, são realizadas comparações entre redes Mux Tree, Clos rearranjáveis de 3 estágios (Clos R3), e Ômega com diferentes arranjos nas portas de saída. A rede Ômega apresenta o sufixo Gi para diferenciar os arranjos das portas de saída. Ômega G1 representa uma rede cuja saída é agrupada no topo, como é mostrado na Figura 21d. A rede Ômega G2 terá disponíveis as saídas {0, 2, 4, 6, ...} (Figura 24b), e a rede Ômega G4 terá disponíveis as saídas {0, 4, 8, 12, 16, ...}.

A taxa média de bloqueio utilizada nas comparações foi determinada pela execução do Algoritmo 2 com 1 milhão de amostras para cada tamanho de rede. Esse número foi determinado de acordo com o método apresentado na seção 3.2. Na Figura 32 são apresentados dois gráficos para as redes de tamanho 2048x32 e 4096x32. No eixo y, é apresentado a taxa média de bloqueio, e no eixo x a quantidade de amostras utilizadas para determinar a taxa. O eixo x está em escala log_{10}, assim o primeiro trecho contém 9 amostras e apresenta uma grande variação na taxa média de bloqueio. O último trecho do gráfico apresenta 900 mil amostras, sendo que a variação da taxa média de bloqueio é ≈ 0. O comportamento é semelhante para outros tamanhos de rede, e no Apêndice E são

apresentados as variações da taxa média de bloqueio em relação à quantidade de amostras para todas os tamanhos de redes utilizados.



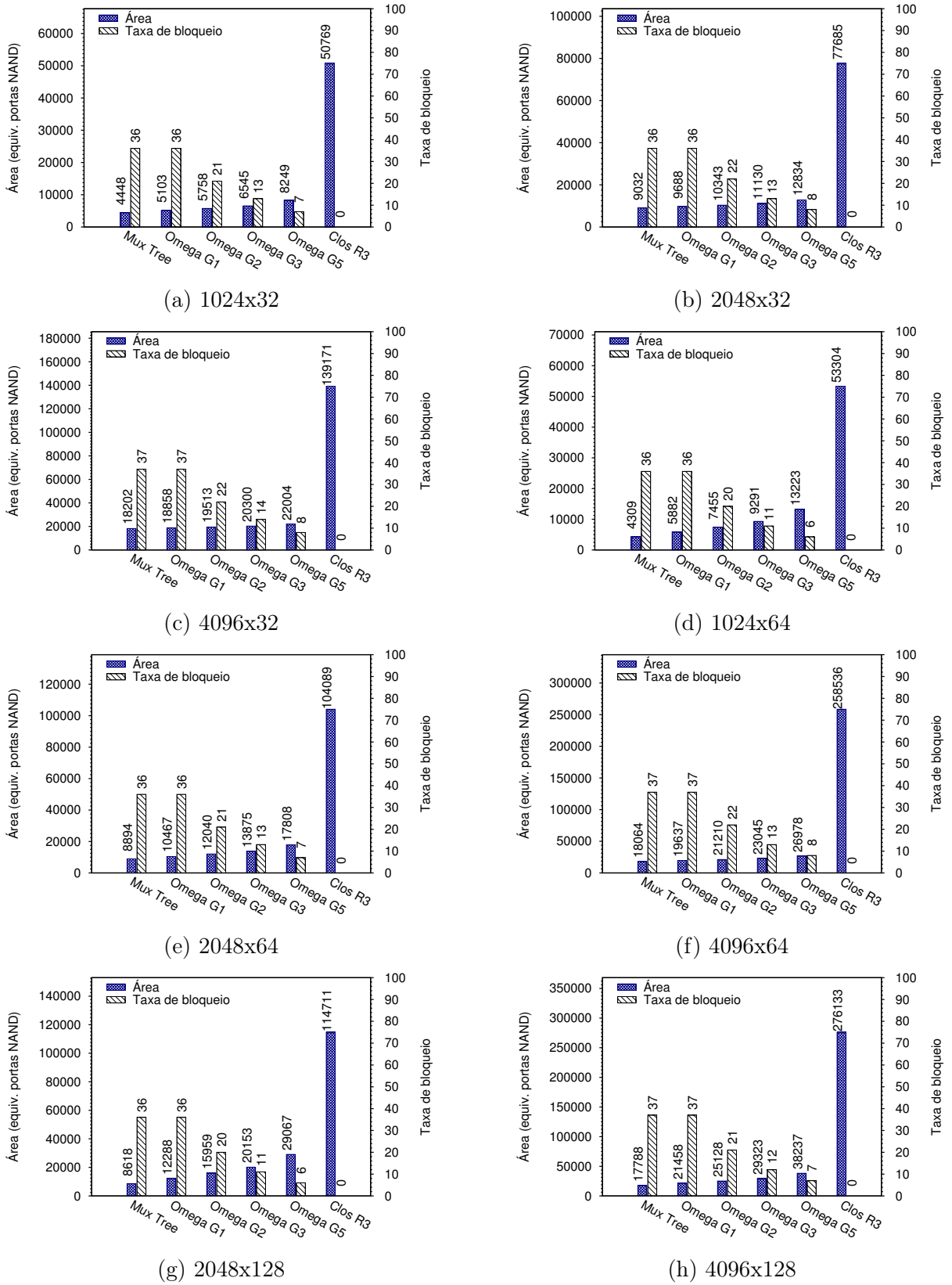
(a) 2048x32 (b) 4096x32

Figura 32 – Quantidade de amostras nos experimentos. No eixo y, os gráficos apresentam a taxa média de bloqueio das redes, e no eixo x a quantidade de amostras para determinar a taxa. O eixo x está em escala log_{10}, e apresenta no último trecho 900 mil amostras.

Na Figura 33 são apresentadas as taxas de bloqueio e as áreas para as redes Mux Tree, Ômega assimétrica e Clos R3[1]. Cada gráfico da Figura 33 contém dois eixos-y. No eixo-y da esquerda está representado os valores computados pela síntese em termos de quantidade de portas NAND equivalentes. O eixo-y da direita representa a taxa de bloqueio computada utilizando o Algoritmo 2. Os resultados apresentam a rede Ômega em 4 diferentes arranjos na saída da rede: G1, G2, G3 e G5. O motivo pelo qual o resultado para rede Ômega G4 não é apresentado é que taxa de bloqueio apresentada decai pouco em relação à rede Ômega G3. Os dados detalhados dos experimentos estão no Apêndice F.

A Figura 33c mostra que a rede Ômega G5 atinge a marca de **4.6 vezes menos bloqueio** em uma rede de 4096x32, ao preço do **elevar o custo de área em 21%** comparado com uma rede Mux Tree do mesmo tamanho. O projetista também pode escolher um custo menor, porém, ainda conseguir uma redução na taxa de bloqueio em relação à Mux Tree. Por exemplo, uma rede Ômega G3 de 4096x32 reduz a taxa de bloqueio em 3 vezes comparado com uma Mux Tree. Esta configuração aumenta o custo de área em 12%, que é um custo menor comparado com a Ômega G5. Outra rede com

[1] No Apêndice H é apresentado um comparativo da quantidade de multiplexadores 2x1 e a taxa de bloqueio das redes Ômega e Mux Tree.



(a) 1024x32

(b) 2048x32

(c) 4096x32

(d) 1024x64

(e) 2048x64

(f) 4096x64

(g) 2048x128

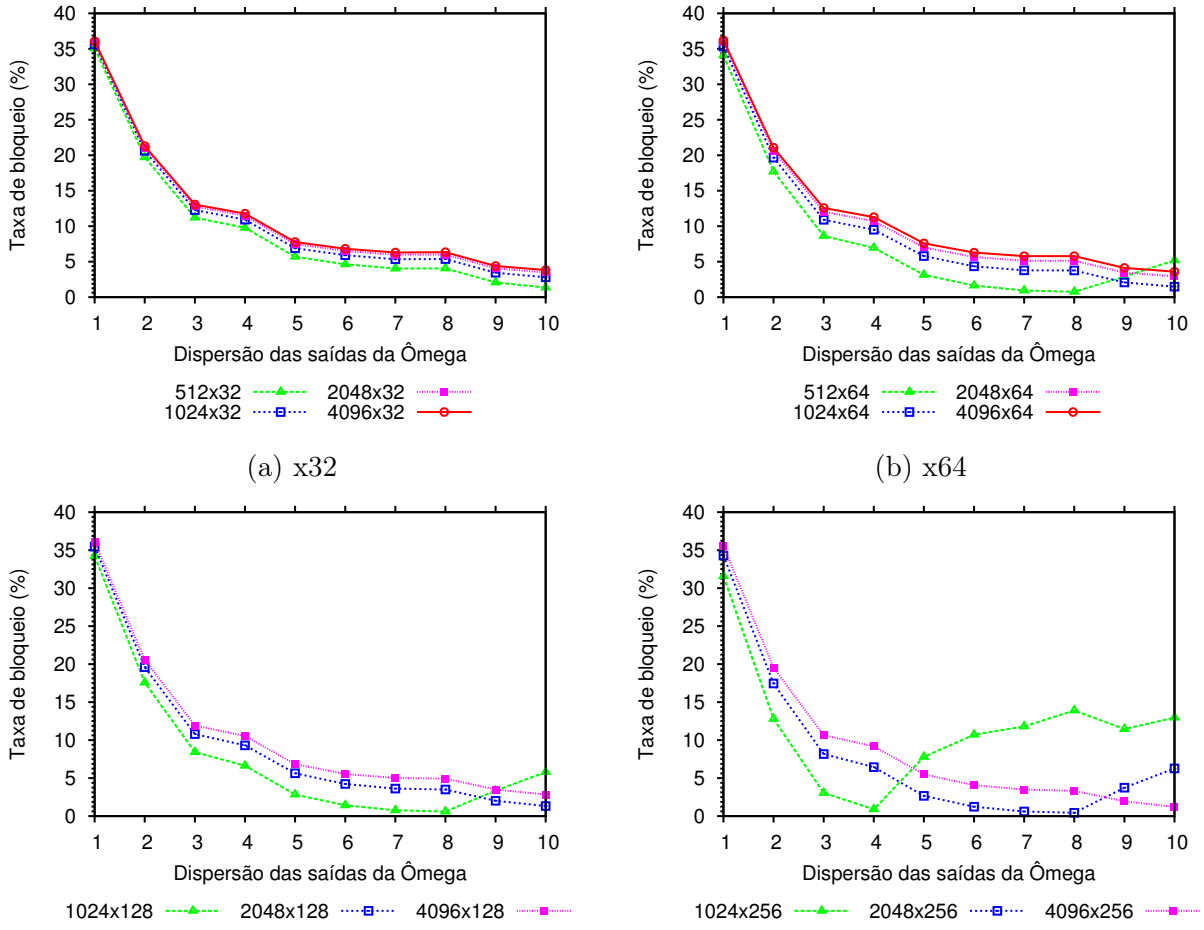
(h) 4096x128

Figura 33 – Resultados de síntese e taxa média de bloqueio. Os gráficos apresentam a comparação das redes Mux Tree, Omega e Clos para vários tamanhos de entrada e saída.

o mesmo tamanho, a Ômega G2, apresenta um pequeno custo de 7%, e também reduz a taxa de bloqueio de 35% para 21%.

Na Figura 33b, o custo adicional imposto pela Ômega G5 2048x32 é ≈ 42% quando comparado com a Mux Tree. Na Figura 33a, a rede Ômega G5 1024x32 apresenta um custo adicional de ≈ 85% comparado com a rede Mux Tree. Para essas redes, a taxa de bloqueio é ≈ 4.5 e ≈ 5.1 vezes menor, respectivamente. Portanto, quanto maior o número de entradas comparado com o número de saídas da rede, menor é o custo apresentado pela rede Ômega, devido ao número de estágios de cada rede.

Na Figura 34 é apresentada a variação da taxa de bloqueio em função do arranjo da saída da rede Ômega. A Figura 34a apresenta curvas semelhantes para os quatro tamanhos de redes: 512x32, 1024x32, 2048x32 e 4096x32. A partir do arranjo 5 (Ômega G5), a curva apresenta uma queda bem suave. Nas Figuras 34b e 34c, as linhas 512x64 e 1024x128 apresentam um aumento na taxa de bloqueio da Ômega G8 para G9 e G10. Este comportamento se deve ao método utilizado para dispersar as saídas da rede. Considere



(a) x32

(b) x64

(c) x128

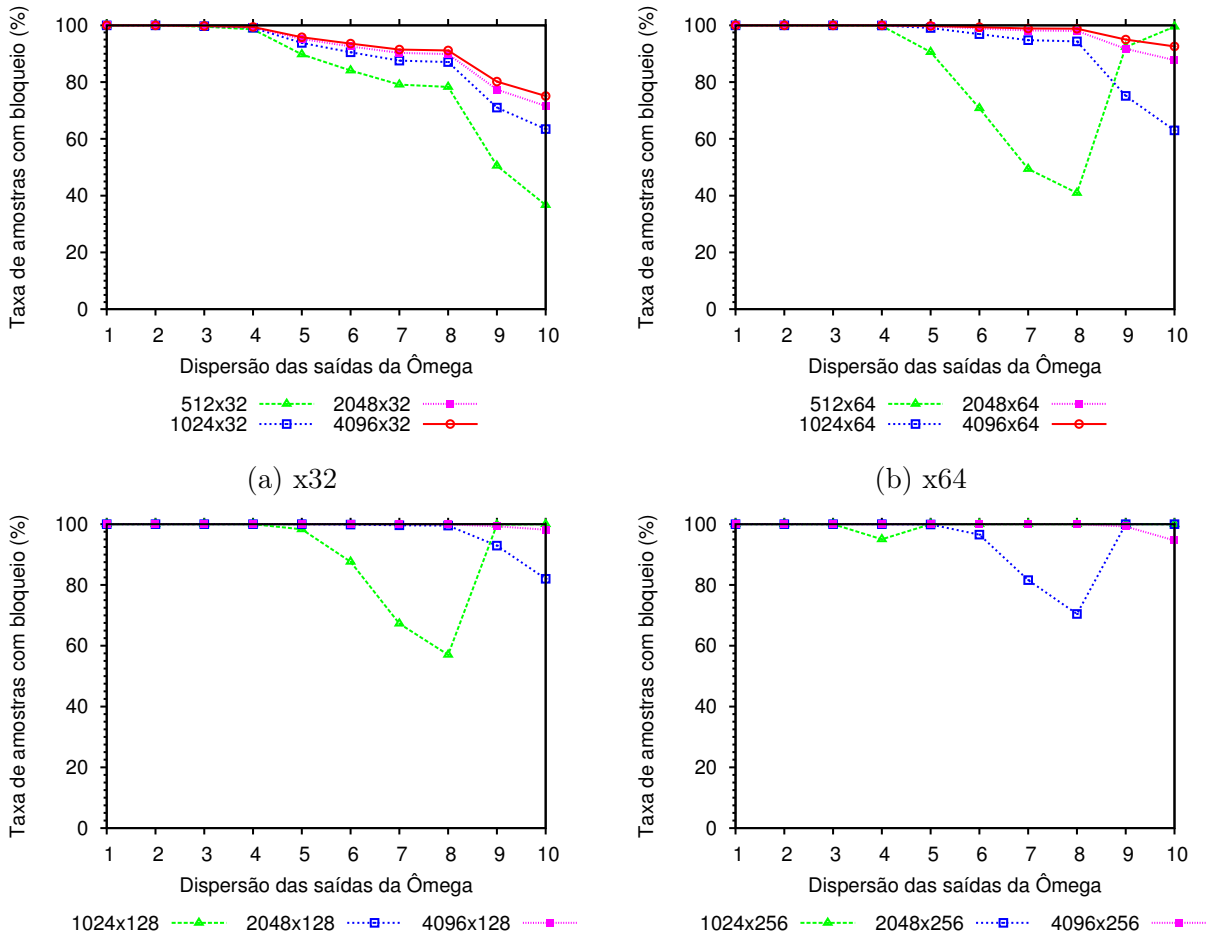
(d) x256

Figura 34 – Comparação da taxa média de bloqueio da rede Ômega variando o arranjo da saída da rede.

por exemplo a rede 512x64; para espalhar de forma uniforme as saídas, o número máximo de espaçamento é $512 \div 64 = 8$. Assim, na Ômega G9 512x64, após espalhar 56 saídas (posição 504), a próxima saída volta para o topo + 1 (posição $504 + 9 \bmod 512 = 1$).

Na Figura 35 é apresentada a taxa de amostras que contém bloqueio em relação às diferentes configurações da rede Ômega assimétrica. Uma amostra é um conjunto de sinais. Se existe pelo menos um sinal do conjunto que foi bloqueado, essa amostra contém bloqueio. Nessa figura pode ser observado que todas as amostras para as redes Ômega G1 contém bloqueio. Quanto menor a diferença entre o número de entradas e saída, mais fácil se torna conseguir uma configuração que não há bloqueio. A Figura 35a mostra que para a rede Ômega G10 512x32 apenas $\approx 37\%$ das amostras apresentam bloqueio, no entanto, essa rede é ≈ 4.8 vezes maior do que uma rede Mux Tree. Também pode ser observado que nesse caso específico, uma rede Crossbar ocupa menos área e não terá bloqueio em nenhuma amostra (ver Tabela 1 no Apêndice F).

Observe que na Figura 34a a taxa de bloqueio para uma rede Ômega G4 4096x32 é



(a) x32

(b) x64

(c) x128

(d) x256

Figura 35 – Comparação da taxa de conjuntos bloqueados da rede Ômega variando o arranjo da saída da rede.

$\approx$ 13%, no entanto pode ser observado na Figura 35a que 99.79% das amostras apresentam bloqueio. Assim, para essa rede, em apenas 0.21% dos casos será possível configurar essa rede sem que haja bloqueio.

Na Figura 36 é apresentada uma comparação da melhor versus a pior configuração obtida nos experimentos. A Mux Tree apresenta um resultado bastante semelhante à rede Ômega G1 em todos os experimentos. Na Figura 36a, o melhor resultado para uma rede Ômega G1, é pior do que o resultado obtido para a Mux Tree. Isto se repete para as Figuras 36b e 36e. Já nas Figuras 36d e 36g, a pior amostra da Ômega é melhor do que a pior amostra da Mux Tree.

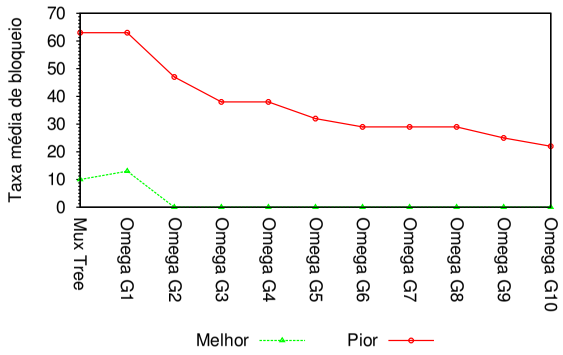
Vale observar que essas diferenças são relativamente pequenas, e a Ômega G1 no geral não apresentam uma boa relação de custo benefício em relação à Mux Tree. No entanto, conforme é apresentado pelas outras redes Ômega com G maior que 1, a rede Ômega apresenta uma boa redução máxima e mínima na taxa de bloqueio. Assim, ao se utilizar uma rede Ômega G5 4096x32, o projetista pode esperar obter um bloqueio máximo de $\approx$ 31%, e em alguns casos pode não ter bloqueio. Em contraste, se utilizar uma rede Mux Tree 4096x32, o bloqueio pode chegar em até $\approx$ 62% e não haverá um caso em que nenhum sinal não é bloqueado.

4.2 Estudo de caso

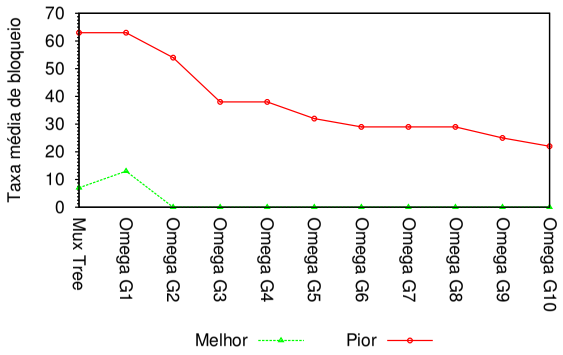
Nas Figuras 37, 38 e 39, são apresentados os comparativos de área para diversas redes de interconexão instanciadas dentro de 3 projetos de código aberto escritos em Verilog. Os projetos foram sintetizados utilizando a ferramenta **Encounter RTL Compiler 11.25** (Cadence Design Systems, 2015) e a área apresentada é em termos de portas NAND equivalentes. Dois dos três projetos são processadores: projeto Amber (SANTIFORT, 2013), e o projeto MIPS 32 Release 1 (AYERS, 2014). O terceiro projeto é o módulo s38584 de *benchmark* do ISCAS (ALBRECHT; Cadence Research Laboratories, 2005). Os detalhamentos dos resultados obtidos nos estudos de caso são apresentados no Apêndice G.

Nesse comparativo, as redes instanciadas são: rede Mux Tree, 3 diferentes redes Ômega, a rede Crossbar, e 3 diferentes redes Clos. Para as redes Ômega, conforme já mencionado no início dessa seção, G seguido de um número representa a forma como a saída da rede foi distribuída. Para as redes Clos, as três redes são redes rearranjáveis de 3, 5 e 7 estágios; respectivamente: Clos R3, Clos R5 e Clos R7.

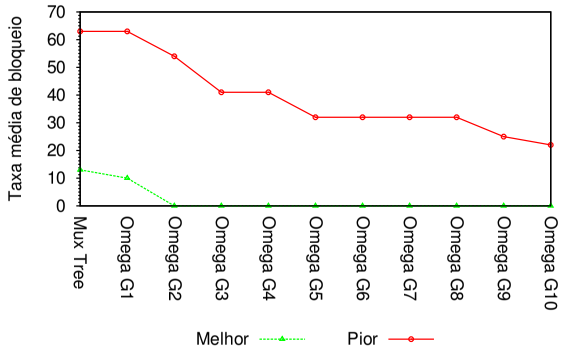
Nas Figuras 37, 38 e 39, pode ser observado que o custo de área da Mux Tree é o menor entre todas as redes. A Figura 37 mostra que o custo adicional para a Mux Tree no projeto Amber é $\approx$ 5% e $\approx$ 14%, respectivamente para as redes de tamanho 100x25 e 256x32. No projeto MIPS 32, o custo adicional é $\approx$ 3% e $\approx$ 8%, respectivamente para as



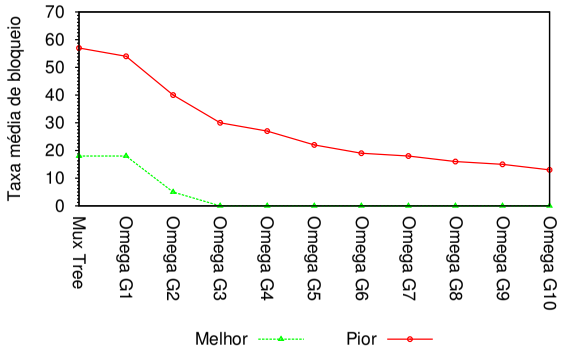
(a) 1024x32



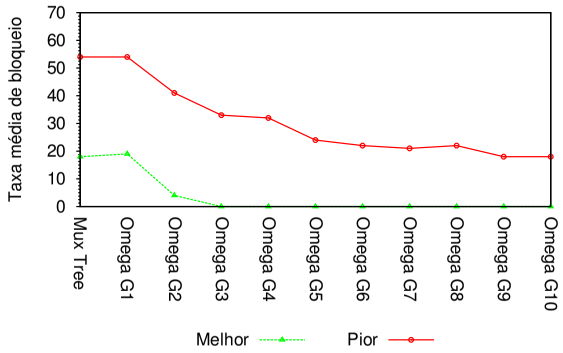
(b) 2048x32



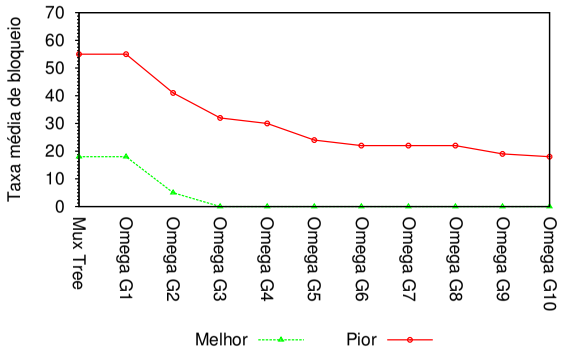
(c) 4096x32



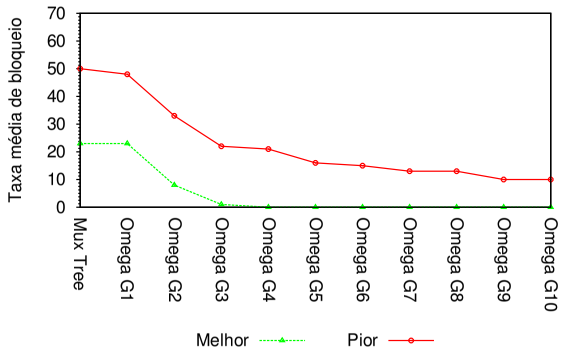
(d) 1024x64



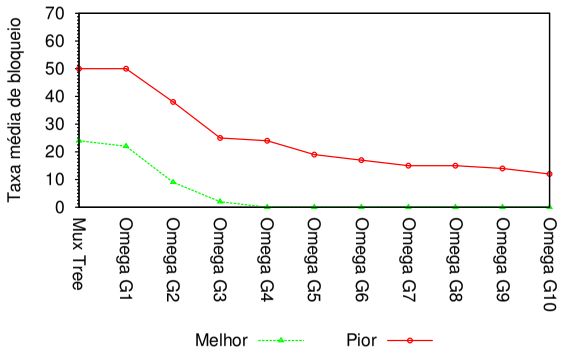
(e) 2048x64



(f) 4096x64



(g) 2048x128



(h) 4096x128

Figura 36 – Comparação da melhor taxa de bloqueio versus a pior taxa de bloqueio para as redes Ômega para vários tamanhos de entrada e saída.

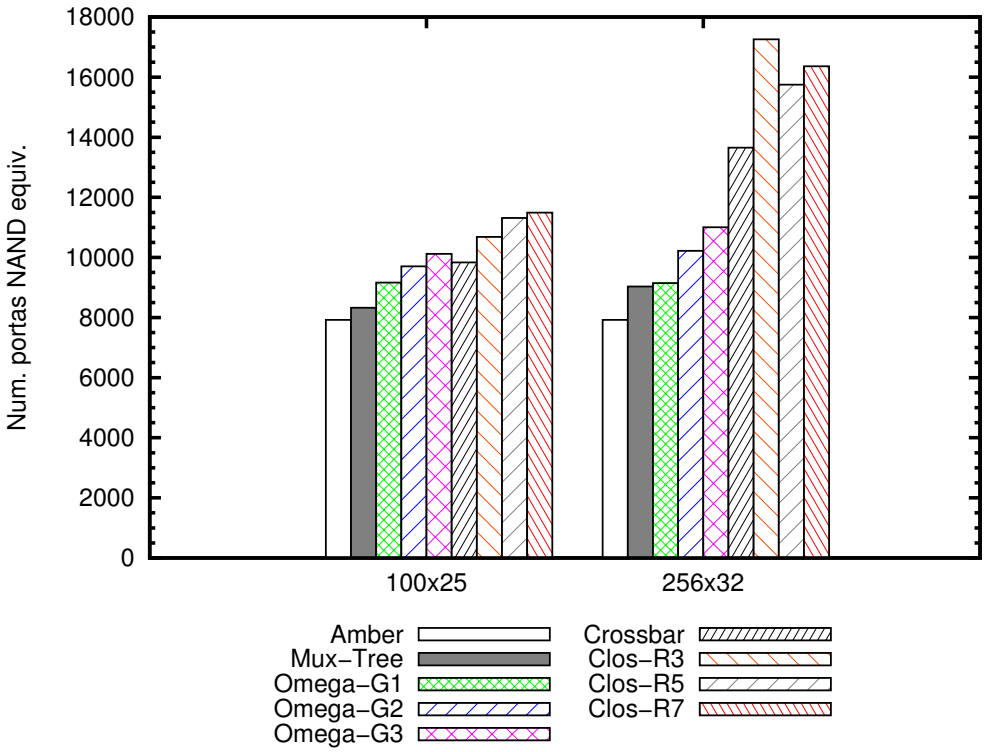


Figura 37 – Comparação de custo de área para as redes instanciadas no projeto Amber.

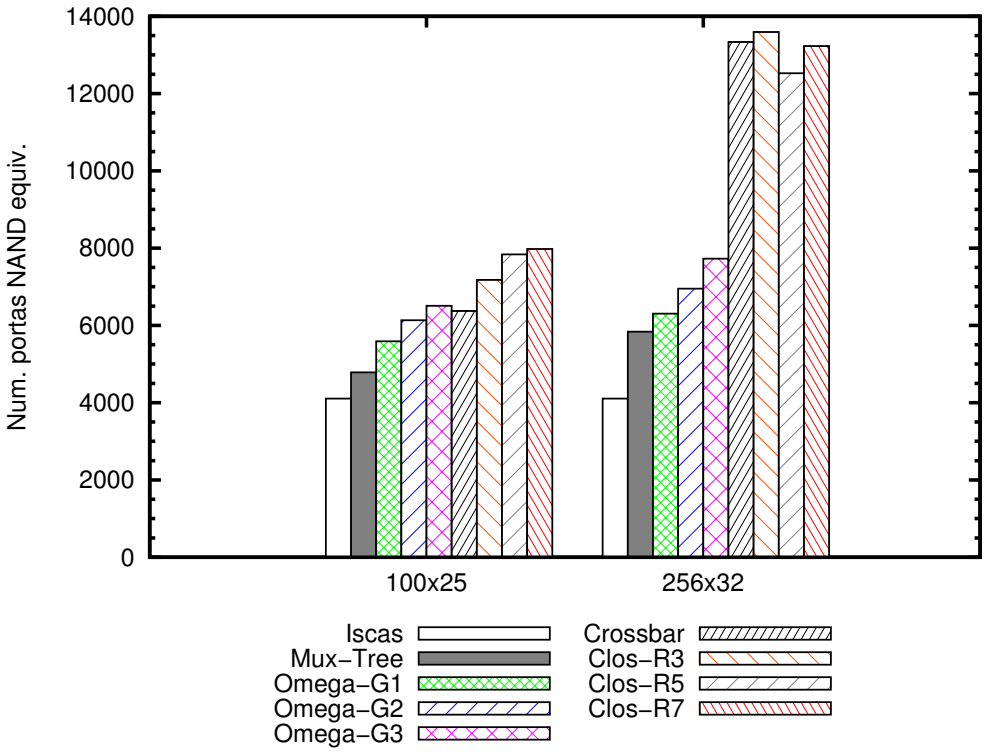


Figura 38 – Comparação de custo de área para as redes instanciadas no projeto ISCAS s38584.

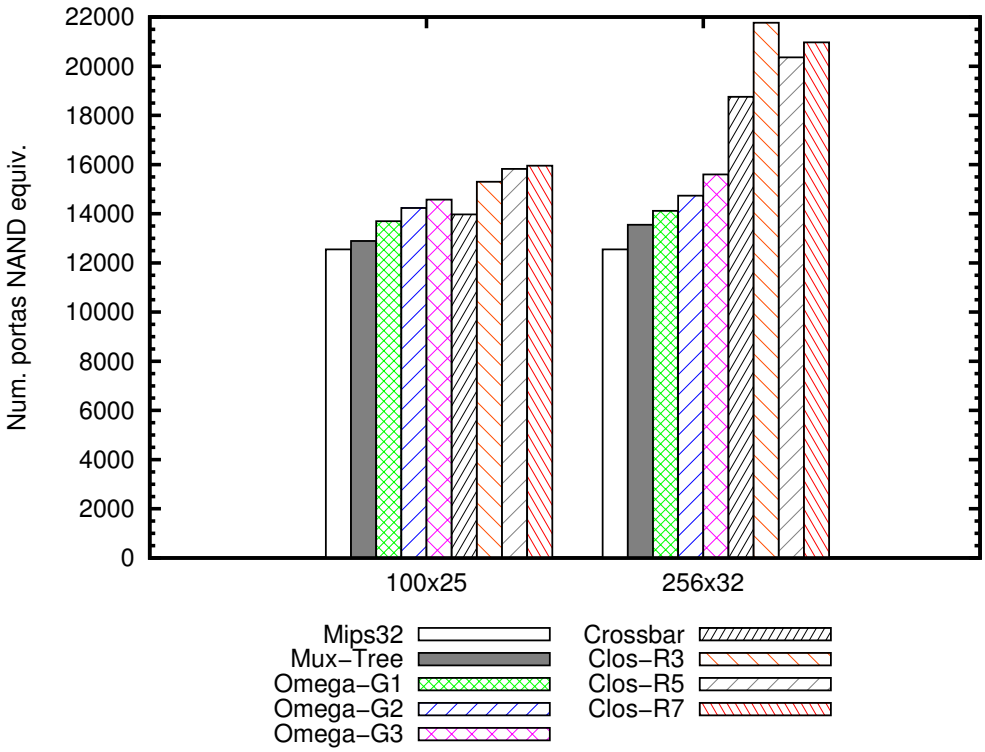


Figura 39 – Comparação de custo de área para as redes instanciadas no projeto Mips 32.

redes de tamanho 100x25 e 256x32. O *benchmark* ISCAS s38584 é um módulo menor que os processadores Amber e MIPS 32, assim, o custo adicional é maior nesse estudo de caso para qualquer rede. Para a Mux Tree, o custo adicional é $\approx 17\%$ e $\approx 42\%$, respectivamente para as redes de tamanho 100x25 e 256x32.

Na Figura 37, a rede Ômega G2 de 100x25 apresenta $\approx 22\%$ de custo em relação ao projeto Amber; a rede Crossbar apresenta $\approx 24\%$ de custo. No entanto, é possível observar que quando a rede se torna maior, como apresentado no comparativo, uma rede 256x32 já apresenta resultados bem diferentes. A rede Crossbar 256x32 instanciada dentro do projeto Amber apresenta um custo de $\approx 72\%$, enquanto a rede Ômega G2 apresenta $\approx 29\%$. A rede Ômega G3 apresenta um custo maior do que a Crossbar em uma rede 100x25. Já na rede de 256x32, uma rede Crossbar é $\approx 24\%$ maior do que a Ômega de mesmo tamanho. Comparando a rede Ômega G2 com a rede Mux Tree, a rede Ômega apresenta um custo de $\approx 11\%$ em relação à rede Mux Tree de mesmo tamanho. No entanto, para essas redes, a Ômega apresenta um uma taxa de bloqueio ≈ 2.7 vezes menor.

As redes Mux Tree apresentam as menores taxas de custo conforme esperado. A rede Clos apresenta maior custo de área em todas as ocasiões, exceto pela rede de 5 estágios instanciada no projeto ISCAS. Esse resultado pode ser explicado pela complexidade do arranjo interno da Clos, que possui mais fios e multiplexadores de diversos tamanhos. Assim, apesar da rede Clos ser composta por menos *crosspoints* do que a rede Crossbar, a rede requer mais área.

Nas redes não-bloqueantes há um crescimento elevado na área quando as redes aumentam de tamanho. Isso pode ser observado nas Figuras 37, 38 e 39. Por exemplo, na Figura 39, o projeto Mips 32 fica $\approx 5\%$ maior ao substituir a rede Mux Tree 100x25 pela mesma rede de 256x32. Ao substituir a rede Ômega G2 de 100x25 por 256x32, a área fica $\approx 3\%$ maior. Ao substituir a rede Crossbar 100x25 por uma de 256x32, a área se torna $\approx 34\%$ maior. Uma rede Clos 100x25 substituída por uma 256x32 torna a área $\approx 42\%$ maior.

5 Conclusão e trabalhos futuros

Nesse trabalho foi proposta uma nova rede de interconexão para depuração pós-silício. Essa rede é uma versão assimétrica da tradicional rede Ômega (LAWRIE, 1975). A rede Ômega assimétrica é construída por um algoritmo que remove os multiplexadores do último estágio de uma rede Ômega convencional. A remoção desses multiplexadores é propagada para os demais estágios até chegar na rede Ômega assimétrica proposta. O processo de remoção é detalhado através de exemplos e apresentado também de forma gráfica.

O trabalho considera que a taxa de bloqueio é a **quantidade de sinais selecionados para monitorar** sobre **a quantidade de sinais roteados** pela rede, e o algoritmo proposto pode gerar diversas versões da rede Ômega assimétrica com diferentes taxas de bloqueio. Dessa forma, em casos de depuração pós-silício com indeterminismo, a rede proposta aumenta a observabilidade do circuito pois permite que projetistas capturem mais valores de sinais que são vitais para a detecção de erros.

Para comparar a rede Ômega assimétrica e a rede Mux Tree, o trabalho apresenta os respectivos algoritmos para calcular a taxa de bloqueio. Os algoritmos apresentados são heurísticas, e dessa forma, o trabalho também apresenta como determinar o número de amostras para o experimento.

O trabalho apresenta a ferramenta Vericonn, capaz de gerar as redes de interconexão Ômega, Mux Tree, Clos, Benes e Crossbar em versões assimétricas em LDH Verilog. A geração automática das redes assimétricas pode ajudar projetistas e companhias a acelerar o processo de depuração. As redes de interconexão multiestágios geradas pelo Vericonn apresentam a mesma interface, e internamente essas redes são compostas por diversas instâncias da rede Crossbar gerada pelo Vericonn. As redes Mux Tree gerados pelo Vericonn podem ser de tamanhos arbitrários.

As documentações de configurações de cada rede gerada pelo Vericonn foram descritos no texto. Os algoritmos utilizados para geração das redes está presente e foi detalhado mostrando a instanciação dos componentes internos e suas conexões. O trabalho apresenta nos apêndices exemplos reais gerado pelo Vericonn, contendo uma simulação, os códigos gerados pela ferramenta e os diagramas esquemáticos.

Vários experimentos foram apresentados comprovando que uma rede Ômega assimétrica pode ser usada para depuração pós-silício, reduzindo a taxa de bloqueio e aumentando a observabilidade do CID. Foram apresentadas através de gráficos, as taxas de bloqueio e as respectivas áreas ocupadas por diversas versões de redes Ômega assimétricas, Mux Trees e redes Clos rearranjáveis. Os resultados demonstram que quanto maior a

diferença entre os tamanhos da entrada e a saída da rede, maior é o benefício na redução de custos de área e taxa de bloqueio. Esse comportamento é compatível com o problema de depuração pós-silício, que possui memória de rastreamento limitada e muitos sinais para explorar.

Outros experimentos mostram as variações da taxa de bloqueio em função das diversas versões da rede Ômega assimétrica proposta. São apresentados comparativos da maior versus a menor taxa de bloqueio para as redes Ômega assimétricas e as redes Mux Trees. Também são apresentados gráficos que relacionam as diversas versões da rede Ômega assimétrica com chances de se obter uma amostra sem bloqueio nas redes. Por fim, estudos de caso apresentam as redes Ômega assimétricas e Mux Trees dentro de projetos de hardware reais.

Alguns tópicos podem ser listados para dar continuidade à esse trabalho. A rede Ômega assimétrica pode ser explorada com outros tamanhos de radix. Conforme apresentado nesse trabalho, a rede Ômega assimétrica possui custo de área menor quando o número de entradas é muito maior do que o número de saídas. Para conseguir outros resultados, um novo trabalho pode utilizar outra forma para construir as redes, ou alterar a estratégia de remoção de multiplexadores.

Em relação à ferramenta, novas redes podem ser incluídas como redes da família *k-ary n-flies*, redes XOR e redes Narasimha. Outros formatos de saída podem ser adicionados ao Vericonn, como VHDL. Novas funcionalidades podem reduzir o tempo gasto na verificação pós-silício, como a instanciação da rede de forma automática dentro de um circuito para depuração, seleção manual e automática dos sinais e inclusão do circuito de armazenamento dos sinais monitorados.

Referências

ABRAMOVICI, M. In-system silicon validation and debug. *Design Test of Computers, IEEE*, v. 25, n. 3, p. 216–223, May 2008. ISSN 0740-7475. Citado 2 vezes nas páginas 3 e 25.

ABRAMOVICI, M. et al. A reconfigurable design-for-debug infrastructure for socs. In: *Design Automation Conference, 2006 43rd ACM/IEEE*. [S.l.: s.n.], 2006. p. 7–12. ISSN 0738-100X. Citado 4 vezes nas páginas 2, 8, 9 e 12.

ALBRECHT, C.; Cadence Research Laboratories. *IWLS 2005 Benchmarks*. 2005. Disponível em: <http://iwls.org/iwls2005/benchmarks.html>. Citado 2 vezes nas páginas 52 e 99.

Altera Corporation. *AN 294: Crosspoint Switch Matrices in MAX II & MAX 3000A Devices*. [S.l.], 2004. Citado na página 39.

ATHAVALE, V. et al. Code coverage of assertions using rtl source code analysis. In: *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*. [S.l.: s.n.], 2014. p. 1–6. Citado na página 12.

AYERS, G. *MIPS32 Release 1*. 2014. Disponível em: <http://opencores.org/project, mips32r1>. Citado 2 vezes nas páginas 52 e 99.

BALSTON, K. et al. Post-silicon code coverage for multiprocessor system-on-chip designs. *Computers, IEEE Transactions on*, v. 62, n. 2, p. 242–246, Feb 2013. ISSN 0018-9340. Citado na página 12.

BASU, K.; MISHRA, P. Rats: Restoration-aware trace signal selection for post-silicon validation. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, v. 21, n. 4, p. 605–613, April 2013. ISSN 1063-8210. Citado na página 2.

Cadence Design Systems. *Encounter RTL Compiler*. 2015. Disponível em: <http: //www.cadence.com/rl/Resources/datasheets/encounter_rtlcompiler.pdf>. Citado 4 vezes nas páginas 47, 52, 91 e 99.

CHANG, K.-H.; MARKOV, I.; BERTACCO, V. Automating post-silicon debugging and repair. In: *Computer-Aided Design, 2007. ICCAD 2007. IEEE/ACM International Conference on*. [S.l.: s.n.], 2007. p. 91–98. ISSN 1092-3152. Citado na página 1.

CLOS, C. A study of non-blocking switching networks. *Bell System Technical Journal*, Blackwell Publishing Ltd, v. 32, n. 2, p. 406–424, 1953. ISSN 1538-7305. Citado 2 vezes nas páginas 15 e 17.

DALLY, W.; TOWLES, B. *Principles and Practices of Interconnection Networks*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003. ISBN 0122007514. Citado 4 vezes nas páginas 13, 19, 21 e 26.

DEORIO, A.; KHUDIA, D. S.; BERTACCO, V. Post-silicon bug diagnosis with inconsistent executions. In: *Proceedings of the International Conference on Computer-Aided Design*. Piscataway, NJ, USA: IEEE Press, 2011. (ICCAD '11), p. 755–761. ISBN 978-1-4577-1398-9. Citado 2 vezes nas páginas 3 e 10.

DUGUID, A. *Structural properties of switching networks*. [S.l.], 1959. Citado na página 19.

FERREIRA, R.; VENDRAMINI, J.; NACIF, M. Dynamic reconfigurable multicast interconnections by using radix-4 multistage networks in fpga. In: *Industrial Informatics (INDIN), 2011 9th IEEE International Conference on*. [S.l.: s.n.], 2011. p. 810–815. Citado na página 23.

FERREIRA, R. S. et al. Fast placement and routing by extending coarse-grained reconfigurable arrays with omega networks. *Journal of Systems Architecture*, v. 57, n. 8, p. 761 – 777, 2011. ISSN 1383-7621. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1383762111000373>. Citado na página 24.

GAO, M.; LISHERNESS, P.; CHENG, K.-T. Post-silicon bug detection for variation induced electrical bugs. In: *Design Automation Conference (ASP-DAC), 2011 16th Asia and South Pacific*. [S.l.: s.n.], 2011. p. 273–278. ISSN 2153-6961. Citado na página 8.

GOKE, L. R.; LIPOVSKI, G. J. Banyan networks for partitioning multiprocessor systems. In: *Proceedings of the 1st Annual Symposium on Computer Architecture*. New York, NY, USA: ACM, 1973. (ISCA '73), p. 21–28. Citado na página 19.

GRAMMATIKAKIS, M.; HSU, D.; KRAETZL, M. *Parallel System Interconnections and Communications*. [S.l.]: Taylor & Francis, 2000. ISBN 9780849331534. Citado 4 vezes nas páginas 16, 23, 24 e 26.

HUNG, E.; WILTON, S. Scalable signal selection for post-silicon debug. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, v. 21, n. 6, p. 1103–1115, June 2013. ISSN 1063-8210. Citado 2 vezes nas páginas 2 e 13.

HUNG, E.; WILTON, S. J. E. On evaluating signal selection algorithms for post-silicon debug. In: *Quality Electronic Design (ISQED), 2011 12th International Symposium on*. [S.l.: s.n.], 2011. p. 1–7. ISSN 1948-3287. Citado na página 2.

HWANG, F. K. *The Mathematical Theory of Nonblocking Switching Networks (Series on Applied Mathematics)*. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 2004. ISBN 9812560424. Citado na página 19.

IBM. *Revenue lost by being late to market*. 2006. Citado 2 vezes nas páginas 1 e 5.

IBM Systems and Technology Group. *CMOS 7RF (CMRF7SF) 1.8V (12-Track) Standard Cell Databook*. [S.l.], 2010. Citado 3 vezes nas páginas 47, 91 e 99.

IEEE Standard for Standard SystemC Language Reference Manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, p. 1–638, Jan 2012. Citado na página 6.

IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, p. 1–560, 2006. Citado na página 6.

IEEE Standard VHDL Language Reference Manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, p. c1–626, Jan 2009. Citado na página 6.

Intel Corporation. *Transistors to Transformations: From sand to circuits - How Intel makes chips*. 2012. Disponível em: <http://exploreintel.com/assets/pdf/aloha/museum-transistors-to-transformations-brochure.pdf>. Citado na página 8.

KERN, C.; GREENSTREET, M. R. Formal verification in hardware design: A survey. *ACM Trans. Des. Autom. Electron. Syst.*, ACM, New York, NY, USA, v. 4, n. 2, p. 123–193, abr. 1999. ISSN 1084-4309. Citado 2 vezes nas páginas 1 e 7.

KIENHUIS, A. C. J. . *Design Space Exploration of Stream-based Dataflow Architectures: Methods and Tools*. PhD thesis — Delft University of Technology, 1999. Disponível em: <http://ptolemy.eecs.berkeley.edu/~kienhuis/ftp/thesis.pdf>. Citado na página 6.

KO, H.; NICOLICI, N. Algorithms for state restoration and trace-signal selection for data acquisition in silicon debug. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, v. 28, n. 2, p. 285–297, Feb 2009. ISSN 0278-0070. Citado 2 vezes nas páginas 2 e 12.

KO, H.; NICOLICI, N. Combining scan and trace buffers for enhancing real-time observability in post-silicon debugging. In: *Test Symposium (ETS), 2010 15th IEEE European*. [S.l.: s.n.], 2010. p. 62–67. ISSN 1530-1877. Citado na página 2.

LAHAUT, D. G. de; GERMAIN, C. Static communications in parallel scientific programs. In: HALATSIS, C. et al. (Ed.). *PARLE'94 Parallel Architectures and Languages Europe*. [S.l.]: Springer Berlin Heidelberg, 1994, (Lecture Notes in Computer Science, v. 817). p. 262–276. ISBN 978-3-540-58184-0. Citado na página 24.

LAWRIE, D. Access and alignment of data in an array processor. *Computers, IEEE Transactions on*, C-24, n. 12, p. 1145–1155, Dec 1975. ISSN 0018-9340. Citado 3 vezes nas páginas 3, 21 e 57.

LIU, X.; XU, Q. Interconnection fabric design for tracing signals in post-silicon validation. In: *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*. [S.l.: s.n.], 2009. p. 352–357. ISSN 0738-100X. Citado na página 27.

LIU, X.; XU, Q. On efficient silicon debug with flexible trace interconnection fabric. In: *Test Conference (ITC), 2012 IEEE International*. [S.l.: s.n.], 2012. p. 1–9. ISSN 1089-3539. Citado na página 28.

LIU, X.; XU, Q. On signal selection for visibility enhancement in trace-based post-silicon validation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, v. 31, n. 8, p. 1263–1274, Aug 2012. ISSN 0278-0070. Citado na página 12.

LIU, X.; XU, Q. *Trace-Based Post-Silicon Validation for VLSI Circuits*. [S.l.]: Springer, 2014. Citado 2 vezes nas páginas 2 e 6.

MARTIN, G.; SMITH, G. High-level synthesis: Past, present, and future. *Design Test of Computers, IEEE*, v. 26, n. 4, p. 18–25, July 2009. ISSN 0740-7475. Citado na página 6.

MITRA, S.; SESHIA, S.; NICOLICI, N. Post-silicon validation opportunities, challenges and recent advances. In: *Design Automation Conference (DAC), 2010 47th ACM/IEEE*. [S.l.: s.n.], 2010. p. 12–17. ISSN 0738-100X. Citado 3 vezes nas páginas 1, 3 e 10.

NACIF, J. A. et al. An assertion library for on-chip white-box verification at run-time. In: *Proceedings of Latin American Test WorkShop*. [S.l.: s.n.], 2003. Citado na página 9.

NACIF, J. A. M. *Processador de asserções para depuração de circuitos integrados em tempo de execução*. Dissertação (Mestrado) — Universidade Federal de Minas Gerais, 2004. Citado na página 9.

NACIF, J. A. M.; PAULA, F. M. de. The chip is ready. am i done? on-chip verification using assertion processors. In: *IFIP/IEEE International Conference on Very Large Scale Integration, 2003. VLSI-SoC 2003*. [S.l.: s.n.], 2003. Citado na página 9.

NARASIMHA, M. A recursive concentrator structure with applications to self-routing switching networks. *Communications, IEEE Transactions on*, v. 42, n. 234, p. 896–898, Feb 1994. ISSN 0090-6778. Citado na página 27.

NARAYANAN, S.; DAS, A. An efficient scheme to diagnose scan chains. In: *Test Conference, 1997. Proceedings., International*. [S.l.: s.n.], 1997. p. 704–713. ISSN 1089-3539. Citado na página 10.

NICOLICI, N.; KO, H. Design-for-debug for post-silicon validation: Can high-level descriptions help? In: *High Level Design Validation and Test Workshop, 2009. HLDVT 2009. IEEE International*. [S.l.: s.n.], 2009. p. 172–175. ISSN 1552-6674. Citado na página 11.

ORUC, A.; ORUC, M. On testing isomorphism of permutation networks. *Computers, IEEE Transactions on*, C-34, n. 10, p. 958–962, Oct 1985. ISSN 0018-9340. Citado na página 19.

PAL, B. et al. Accelerating assertion coverage with adaptive testbenches. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, v. 27, n. 5, p. 967–972, May 2008. ISSN 0278-0070. Citado na página 12.

PATEL, J. H. Processor-memory interconnections for multiprocessors. In: *Proceedings of the 6th Annual Symposium on Computer Architecture*. New York, NY, USA: ACM, 1979. (ISCA '79), p. 168–177. Citado na página 19.

PEASE, M. C. The indirect binary n-cube microprocessor array. *Computers, IEEE Transactions on*, C-26, n. 5, p. 458–473, May 1977. ISSN 0018-9340. Citado na página 15.

PRABHAKAR, S.; HSIAO, M. Using non-trivial logic implications for trace buffer-based silicon debug. In: *Asian Test Symposium, 2009. ATS '09.* [S.l.: s.n.], 2009. p. 131–136. ISSN 1081-7735. Citado na página 13.

PRABHAKAR, S.; HSIAO, M. Multiplexed trace signal selection using non-trivial implication-based correlation. In: *Quality Electronic Design (ISQED), 2010 11th International Symposium on*. [S.l.: s.n.], 2010. p. 697–704. ISSN 1948-3287. Citado na página 28.

QUINTON, B.; WILTON, S. Concentrator access networks for programmable logic cores on socs. In: *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*. [S.l.: s.n.], 2005. p. 45–48 Vol. 1. Citado na página 27.

ROOTSELAAR, G. van; VERMEULEN, B. Silicon debug: scan chains alone are not enough. In: *Test Conference, 1999. Proceedings. International*. [S.l.: s.n.], 1999. p. 892–902. ISSN 1089-3539. Citado 2 vezes nas páginas 7 e 8.

ROSINGER, P.; AL-HASHIMI, B.; NICOLICI, N. Scan architecture with mutually exclusive scan segment activation for shift- and capture-power reduction. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, v. 23, n. 7, p. 1142–1153, July 2004. ISSN 0278-0070. Citado na página 10.

SANGUINETTI, J.; ZHANG, E. The relationship of code coverage metrics on high-level and rtl code. In: *High Level Design Validation and Test Workshop (HLDVT), 2010 IEEE International*. [S.l.: s.n.], 2010. p. 138–141. ISSN 1552-6674. Citado na página 12.

SANTIFORT, C. *Amber Project User Guide*. 2013. Disponível em: <http://opencores.org/websvn,filedetails?repname=amber&path=%2Famber%2Ftrunk%2Fdoc%2Famber-user-guide.pdf>. Citado 2 vezes nas páginas 52 e 99.

SIBAL, S.; ZHANG, J. On a class of banyan networks and tandem banyan switching fabrics. *Communications, IEEE Transactions on*, v. 43, n. 7, p. 2231–2240, Jul 1995. ISSN 0090-6778. Citado na página 20.

SIEGEL, H. Interconnection networks for simd machines. *Computer*, v. 12, n. 6, p. 57–65, June 1979. ISSN 0018-9162. Citado na página 23.

SLEPIAN, D. Two theorems on a particular crossbar switching network. Unpublished manuscript. 1952. Citado na página 19.

SRIDHAR, M. A fast algorithm for testing isomorphism of permutation networks. *Computers, IEEE Transactions on*, v. 38, n. 6, p. 903–909, Jun 1989. ISSN 0018-9340. Citado na página 19.

STONE, H. S. Parallel processing with the perfect shuffle. *Computers, IEEE Transactions on*, C-20, n. 2, p. 153–161, Feb 1971. ISSN 0018-9340. Citado na página 22.

TALUPUR, M. Hardware model checking: Status, challenges, and opportunities. In: *Formal Methods in Computer-Aided Design (FMCAD), 2011*. [S.l.: s.n.], 2011. p. 154–154. Citado na página 7.

VENDRAMINI, J.; FERREIRA, R. Parallel routing algorithm for extra level omega networks on reconfigurable systems. In: *Computing Systems (WSCAD-SCC), 2010 11th Symposium on*. [S.l.: s.n.], 2010. p. 1–8. Citado na página 25.

VERMEULEN, B.; GOEL, S. Design for debug: catching design errors in digital chips. *Design Test of Computers, IEEE*, v. 19, n. 3, p. 35–43, May 2002. ISSN 0740-7475. Citado 2 vezes nas páginas 1 e 8.

VERMEULEN, B.; GOOSSENS, K. *Debugging Systems-on-Chip: Communication-centric and Abstraction-based Techniques*. [S.l.]: Springer, 2014. (Embedded Systems). ISBN 9783319062426. Citado 3 vezes nas páginas 1, 5 e 7.

WU, C. lin; FENG, T.-Y. On a class of multistage interconnection networks. *Computers, IEEE Transactions on*, C-29, n. 8, p. 694–702, Aug 1980. ISSN 0018-9340. Citado na página 21.

Xilinx Inc. *Vivado Design Suite 2014, Release Notes UG973 (v2014.2)*. 2014. Disponível em: <http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_2/ug973-vivado-release-notes-install-license.pdf>. Citado 7 vezes nas páginas 67, 68, 70, 71, 76, 81 e 84.

YANG, J.-S.; TOUBA, N. Enhancing silicon debug via periodic monitoring. In: *Defect and Fault Tolerance of VLSI Systems, 2008. DFTVS '08. IEEE International Symposium on*. [S.l.: s.n.], 2008. p. 125–133. ISSN 1550-5774. Citado na página 28.

YANG, J.-S.; TOUBA, N. Automated selection of signals to observe for efficient silicon debug. In: *VLSI Test Symposium, 2009. VTS '09. 27th IEEE*. [S.l.: s.n.], 2009. p. 79–84. ISSN 1093-0167. Citado na página 13.

YANG, S.; WILLE, R.; DRECHSLER, R. Determining cases of scenarios to improve coverage in simulation-based verification. In: *Integrated Circuits and Systems Design (SBCCI), 2014 27th Symposium on*. [S.l.: s.n.], 2014. p. 1–7. Citado na página 7.

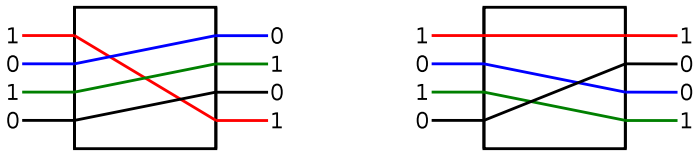
YANG, Y.-S. et al. Automated silicon debug data analysis techniques for a hardware data acquisition environment. In: *Quality Electronic Design (ISQED), 2010 11th International Symposium on*. [S.l.: s.n.], 2010. p. 675–682. ISSN 1948-3287. Citado na página 13.

Apêndices

APÊNDICE A – Exemplo de Crossbar (4x4) gerada pelo Vericonn

Nesse apêndice é apresentado um exemplo de configuração de uma rede Crossbar 4x4 gerada pelo Vericonn. São apresentados os códigos gerados e o gráfico de onda da simulação. A arquitetura da rede Crossbar é apresentada na seção 3.3.1.

Para demonstrar o processo de configuração do módulo Crossbar, esse apêndice apresenta a configuração de duas rotas para uma rede Crossbar 4x4, conforme é definido na Figura 40. O dado de entrada 1010_b é roteado de duas formas diferentes e apresenta duas saídas diferentes. A Figura 41 apresenta a simulação para configurar essas rotas no módulo DSM4x4. Na Figura 41, os valores da entrada (**di**) vão para a saída (**do**) de acordo com a configuração presente nos endereços de entrada e saída ($in\_add \rightarrow out\_add$). Na primeira configuração, os bits da entrada (**di**) são conectados à saída da seguinte forma: $0 \rightarrow 3$, $1 \rightarrow 0$, $2 \rightarrow 1$ e $3 \rightarrow 2$. A saída (**do**) resultante é: 1010_b. Na segunda configuração, $0 \rightarrow 0$, $1 \rightarrow 2$, $2 \rightarrow 3$ e $3 \rightarrow 1$, tornam a saída (do) 1001_b. O Código 1 contém o código da simulação em Verilog.



(a) Rota 1. (b) Rota 2.

Figura 40 – Na Figura (a), a entrada 1010_b é roteada conectando $0 \rightarrow 3$, $1 \rightarrow 0$, $2 \rightarrow 1$ e $3 \rightarrow 2$, e apresenta a saída 0101_b. Na Figura (b), a mesma entrada é roteada conectando $0 \rightarrow 0$, $1 \rightarrow 2$, $2 \rightarrow 3$ e $3 \rightarrow 0$, e apresenta a saída 1001_b.

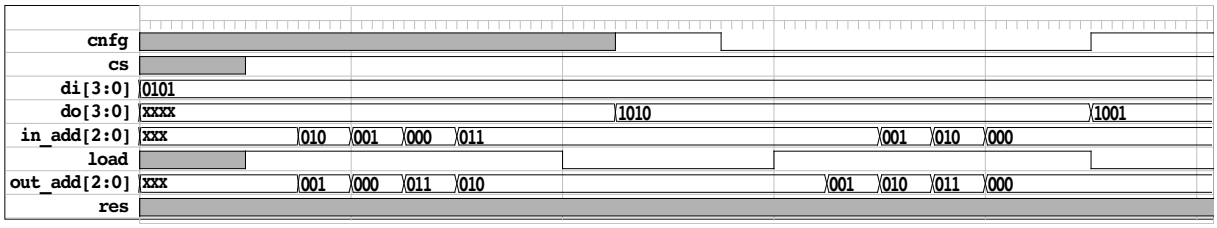


Figura 41 – Simulação para configuração de um módulo Crossbar 4x4 com load/config. O módulo é configurado duas vezes. Quando *load* está alto, os endereços nos barramentos in_add e out_add são gravados. Quando o sinal cnfg está alto, o saída do é alterada de acordo com o valor dos registradores de configuração.

```
module DSM4x4_test;
    reg cnfg, res, load, cs;
    reg [2:0] in_add; reg [2:0] out_add;
    reg [3:0] di; wire [3:0] do;

    DSM4x4 dsm (.cnfg(cnfg), .res(res), .load(load), .cs(cs),
        .in_add(in_add), .out_add(out_add), .di(di), .do(do));

    initial
    begin
        $dumpfile("DSM4x4_test.vcd"); $dumpvars; #22 $finish;

        #0 di = 4'b0101;
        #2 load = 1; #0 cs =   1;

        #1 in_add = 3'b010; #0 out_add = 3'b001;
        #1 in_add = 3'b001; #0 out_add = 3'b000;
        #1 in_add = 3'b000; #0 out_add = 3'b011;
        #1 in_add = 3'b011; #0 out_add = 3'b010;

        #2 load = 0; #1 cnfg = 1;
        #2 cnfg = 0; #1 load = 1;

        #1 in_add = 3'b011; #0 out_add = 3'b001;
        #1 in_add = 3'b001; #0 out_add = 3'b010;
        #1 in_add = 3'b010; #0 out_add = 3'b011;
        #1 in_add = 3'b000; #0 out_add = 3'b000;

        #2 cnfg = 1; #0 load = 0;
    end
endmodule
```

Código 1 – Módulo DSM4x4Test em Verilog

Conforme é apresentado na seção 3.3.1, a arquitetura do módulo Crossbar é composto por 4 módulos internos. O módulo IOCell encapsula um multiplexador que recebe os dados de entrada por um barramento e tem como saída um único sinal. Esse sinal é contém apenas um dos valores do barramento de entrada e é selecionado por um barramento de configuração. Na Figura 28, o funcionamento interno da **Matriz de Switches** é mostrada pelo círculo. Cada multiplexador desse círculo é um módulo IOCell. Na Figura 42 é apresentado o esquemático do módulo IOCell4 com 4 entradas e 1 saída. O esquemático foi gerada pela ferramenta **Xilinx Vivado 2014** (Xilinx Inc., 2014). O código fonte gerado pelo Vericonn para esse módulo é apresentado no Código 2.

```
module IOCell4 (input [3:0] in, input [2:0] sel, output out
);
  reg mux_out;

  always @*
  begin
    case (sel)
      3'b000: mux_out = in[0];
      3'b001: mux_out = in[1];
      3'b010: mux_out = in[2];
      3'b011: mux_out = in[3];
      default: mux_out = 1'bx;
    endcase
  end
  assign out = mux_out;
endmodule
```

Código 2 – Módulo IOCell4 em Verilog

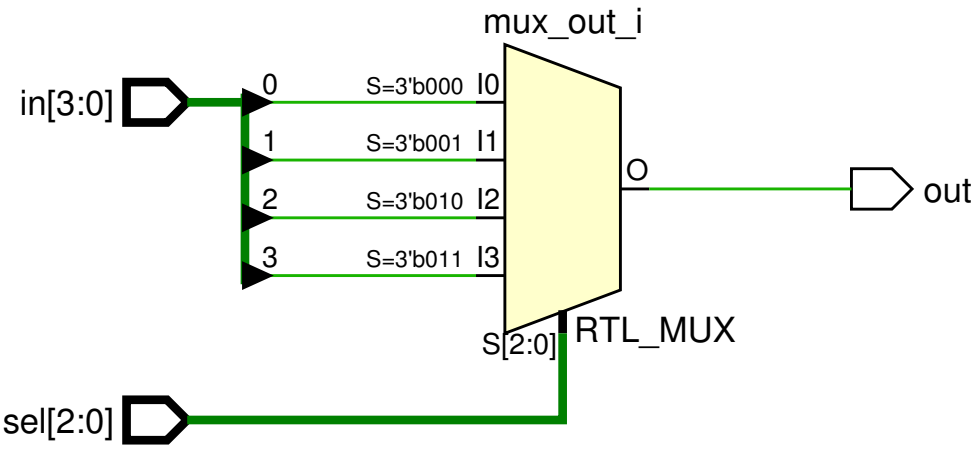


Figura 42 – Esquemático para o módulo IOCell4

Para criar um módulo Crossbar com 4 entradas e 4 saída, é necessário que haja 4 módulos IOCell4. O código gerado pelo Vericonn parar o módulo da Matriz de Switches de 4 entrada e 4 saídas é nomeado SwitchMatrix4x4 e é apresentado no Código 3. O esquemático do módulo é apresentado na Figura 43 (gerada pela ferramenta **Xilinx Vivado 2014** (Xilinx Inc., 2014)). O sinal de entrada é conectado em todos os módulos IOCell, e a saída é a concatenação das saídas de cada IOCell. O módulo SwitchMatrix também tem como entrada um barramento de configuração para cada IOCell.

```
module SwitchMatrix4x4(input [3:0] in, input [2:0] sel0, input [2:0] sel1,
  input [2:0] sel2, input [2:0] sel3, output [3:0] out
);
  IOCell4 output0 (.in(in), .sel(sel0), .out(out[0]));
  IOCell4 output1 (.in(in), .sel(sel1), .out(out[1]));
  IOCell4 output2 (.in(in), .sel(sel2), .out(out[2]));
  IOCell4 output3 (.in(in), .sel(sel3), .out(out[3]));
endmodule
```

Código 3 – Módulo SwitchMatrix4x4 em Verilog

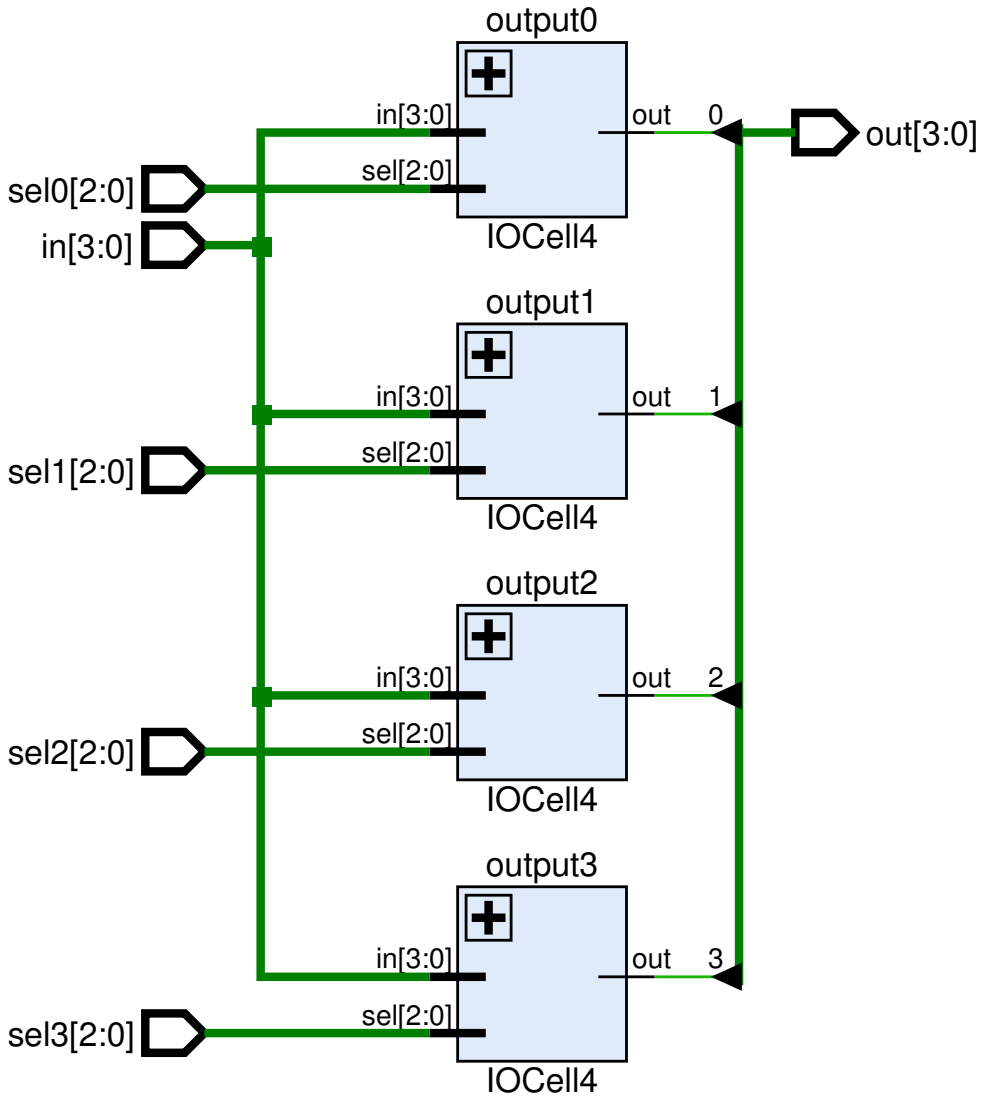


Figura 43 – Esquemático para o módulo SwitchMatrix4x4

Na Figura 28, um módulo chamado **Configuração de Registradores** é responsável por manter a configuração da rede Crossbar. O módulo recebe os endereços de entrada e saída para configurar um único registrador por vez. No módulo gerado pelo Vericonn,

não há um encapsulamento de um módulo de configuração de registradores. A ferramenta gera diversos módulos com o nome Switch, sendo um módulo para configurar cada IOCell do módulo SwitchMatrix. Uma instância do módulo Switch para um Crossbar 4x4 é apresentado no Código 4. O esquemático do módulo é apresentado na Figura 44 (gerada pela ferramenta **Xilinx Vivado 2014** (Xilinx Inc., 2014)). O módulo apresenta o esquema de *load/config*. No modo *load*, um banco de registradores grava as configurações quando **load_enable** está alto. Quando **conf_en** está alto, a configuração é aplicada na rede.

```
module Switch4(input conf_en, input load_en, input [2:0] in_add,
  input out_sel, input reset, output reg [2:0] inadd
);
  reg [2:0] load_inadd_reg; wire load_enable;
  assign load_enable = load_en & out_sel;

  always @(posedge load_enable or posedge reset)
  begin
    if (reset) load_inadd_reg = 3'b0;
    else load_inadd_reg = in_add;
  end

  always @(posedge conf_en or posedge reset)
  begin
    if (reset) inadd = 3'b0;
    else inadd = load_inadd_reg;
  end
endmodule
```

Código 4 – Módulo Switch4 em Verilog

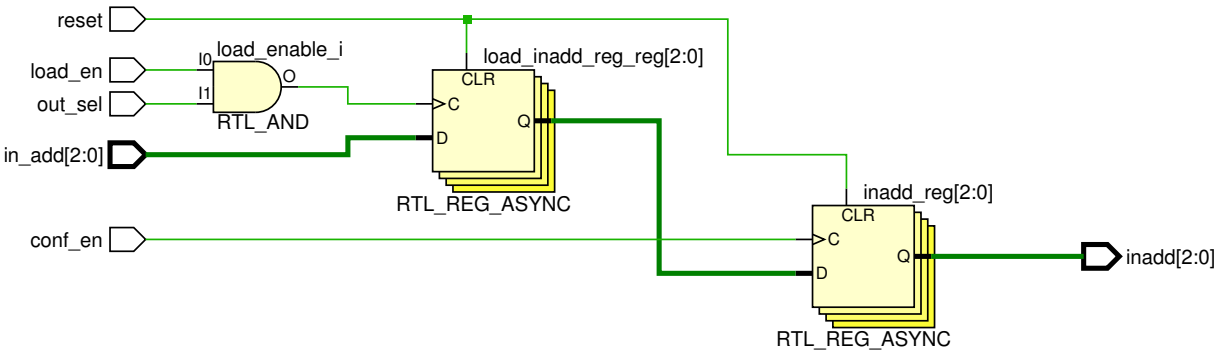


Figura 44 – Esquemático para o módulo Switch4

Na Figura 28 um decodificador é utilizado para traduzir o endereço de saída, que então é enviado para módulo de configuração de registradores. O módulo **Decoder** é gerado pelo Vericonn para realizar essa tradução do endereço numérico sequencial, para

um formato que habilita ou desabilita a configuração de um determinado módulo Switch. A Figura 45 apresenta o esquemático do módulo Decoder4 gerado pela ferramenta **Xilinx Vivado 2014** (Xilinx Inc., 2014). Note que o esquemático em nível de RTL é apenas traduzido para um elemento RTL_ROM. Após a síntese esse RTL_ROM é novamente traduzido para outros elementos (*gates*).

```
module Decoder4(input [2:0] in, output reg [3:0] out
);
  always @*
    case (in)
      3'b000 : out = 4'd1;
      3'b001 : out = 4'd2;
      3'b010 : out = 4'd4;
      3'b011 : out = 4'd8;
      default: out = 4'd0;
    endcase
endmodule
```

Código 5 – Módulo Decoder4 em Verilog

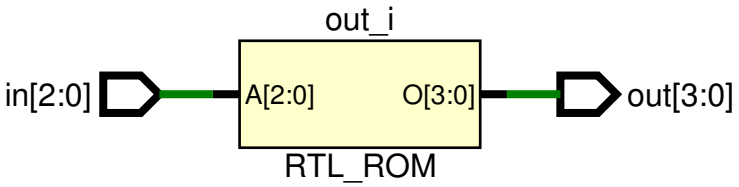


Figura 45 – Esquemático para o módulo Decoder4

O módulo da rede Crossbar é chamado de DSM, e um módulo com 4 entradas e 4 saídas é chamado de DSM4x4. O Código 6 foi gerado pelo Vericonn. Esse módulo instancia um decodificador de endereços de saída, Decoder4, uma matrix de multiplexadores, SwitchMatrix4x4 e 4 módulos Switch4. A conexão de cada módulo é apresentada na Figura 46.

```
module DSM4x4(input cnfg, input res, input load, input cs,
  input [2:0] in_add, input [2:0] out_add,
  input [3:0] di, output [3:0] do
);
  wire [3:0] dec_add;
  wire [2:0] sel0; wire [2:0] sel1;
  wire [2:0] sel2; wire [2:0] sel3;
  wire conf_en; wire load_en; wire reset;

  assign conf_en = cnfg & cs;
  assign load_en = load & cs;
  assign reset = res & cs;

  Decoder4 dec_inst(.in(out_add), .out(dec_add));
  SwitchMatrix4x4 sm_inst(.in(di), .sel0(sel0), .sel1(sel1),
  .sel2(sel2), .sel3(sel3), .out(do));

  Switch4 sw_inst0(.conf_en(conf_en), .load_en(load_en),
  .out_sel(dec_add[0]), .in_add(in_add), .reset(reset), .inadd(sel0));

  Switch4 sw_inst1(.conf_en(conf_en), .load_en(load_en),
  .out_sel(dec_add[1]), .in_add(in_add), .reset(reset), .inadd(sel1));

  Switch4 sw_inst2(.conf_en(conf_en), .load_en(load_en),
  .out_sel(dec_add[2]), .in_add(in_add), .reset(reset), .inadd(sel2));

  Switch4 sw_inst3(.conf_en(conf_en), .load_en(load_en),
  .out_sel(dec_add[3]), .in_add(in_add), .reset(reset), .inadd(sel3));
endmodule
```

Código 6 – Módulo DSM4x4 em Verilog

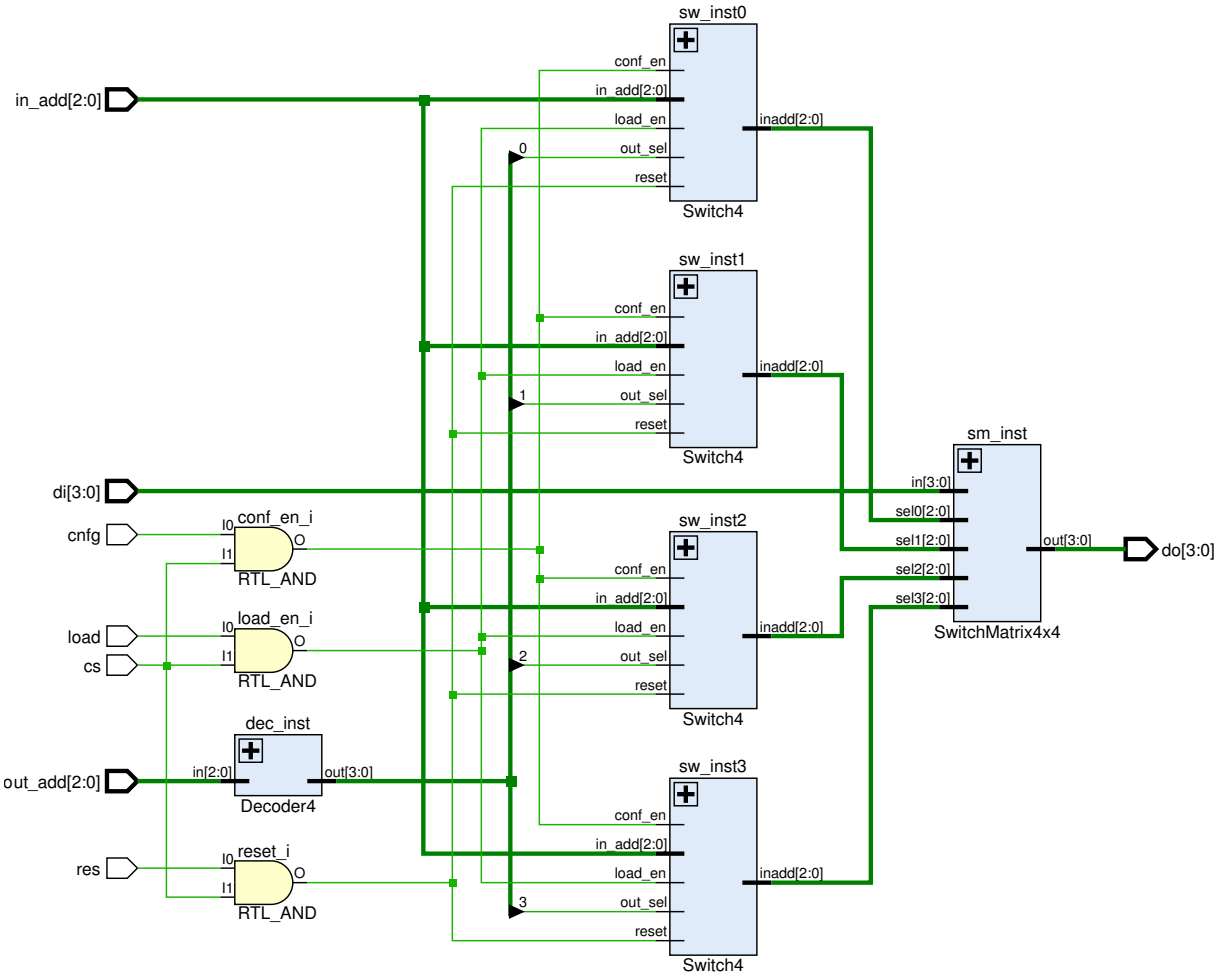


Figura 46 – Esquemático para o módulo DSM4x4

APÊNDICE B – Exemplo de Clos (16x4) gerada pelo Vericonn

Nesse apêndice é apresentado um exemplo de configuração de uma rede Clos 16x4 gerada pelo Vericonn. São apresentados os códigos gerados e o gráfico de onda da simulação. A arquitetura da rede Clos é apresentada na seção 3.3.2.1.

Na Figura 47, $N = 16$ sinais de entrada, $n_1 = 16^{2/(3+1)} = 4$ sinais de entrada em cada Crossbar no primeiro estágio. $M = 4$ sinais de saída, $n_2 = 4^{2/(3+1)} = 2$ sinais de saída do Crossbar no último estágio.

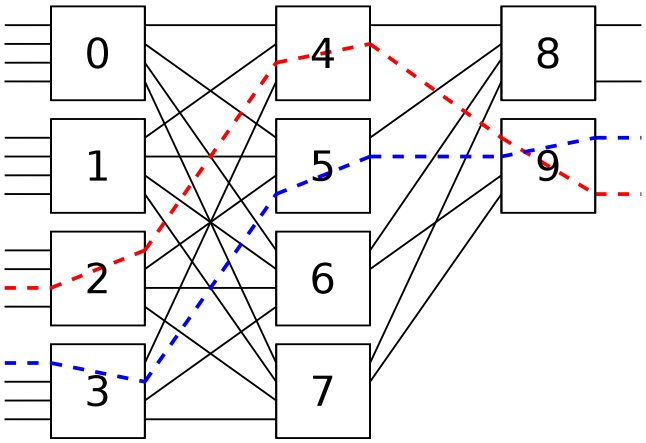


Figura 47 – Roteando 12 → 2 e 10 → 3 na rede Clos 16x4. Para configurar a rota, cada Crossbar interno é configurado separadamente, um a um. Nesse exemplo, 6 tempos de *clock* são necessários: 2 rotas * 3 estágios.

Duas rotas são apresentas na Figura 47. Na arquitetura proposta, para configurar essas rotas é necessário configurar cada Crossbar individualmente. Na Figura 48 é apresentada a simulação para configurar esse módulo da rede Clos de tamanho 16x4. No exemplo, a porta de entrada 10 é conectada à porta de saída 3. A porta de entrada 12 é conectada à porta de saída 2. A tradução de endereço foi desabilitada no barramento **Seleciona Chip**, portanto, cada sinal está conectado à uma rede Crossbar.

Na simulação presente na Figura 48, os ciclos 3, 4, 5 configuram os *switches* 2, 4, 9, e é correspondente à rota vermelha na Figura 47. No ciclo 3, a porta de entrada 2 é conectada à saída 0 do *switch* 2. No ciclo 4, a porta de entrada 2 é conectada à saída 0 do *switch* 4. No ciclo 5, a porta de entrada 0 é conectada à porta de saída 1 do *switch* 9. O bit 10 da entrada da rede tem o valor 1, assim, no ciclo 5 é possível verificar que a saída apresenta o valor 1 na saída 3. O Código 7 apresenta o código da simulação em Verilog.

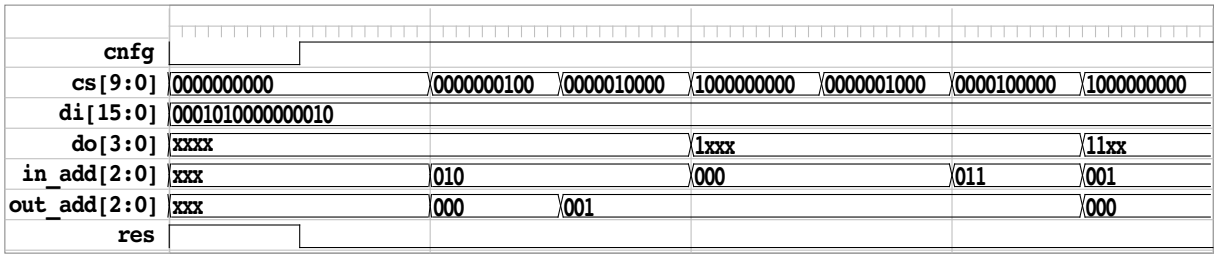


Figura 48 – Simulação para configurar uma rede Clos 16x4. Os bits 10 e 12 do barramento **di** passam por 3 *switches* até a saída. Nos ciclos 3, 4 e 5 são configurados os *switches* da rota vermelha: {2, 4, 9}. Em seguida, nos ciclos 6, 7 e 8, os *switches* da rota azul: {3, 5, 9}.

```
module Clos16x4_test;
  reg cnfg; reg res; reg [9:0] cs;
  reg [2:0] in_add; reg [2:0] out_add;
  reg [15:0] di; wire [3:0] do;

  Clos16x4 clos(.cnfg(cnfg), .res(res),  .cs(cs),
  .in_add(in_add), .out_add(out_add), .di(di), .do(do));

  initial
  begin
    #0 cs = 10'b0000000000; #0 di = 16'b0001010000000010;
    #0 res = 1;  #0 cnfg = 0; #1 cnfg = 1; #0 res = 0;

    // rota vermelha
    #1 cs = 10'b0000000100; #0 in_add = 3'b010; #0 out_add = 3'b000;
    #1 cs = 10'b0000010000; #0 in_add = 3'b010; #0 out_add = 3'b001;
    #1 cs = 10'b1000000000; #0 in_add = 3'b000; #0 out_add = 3'b001;
    // rota azul
    #1 cs = 10'b0000001000; #0 in_add = 3'b000; #0 out_add = 3'b001;
    #1 cs = 10'b0000100000; #0 in_add = 3'b011; #0 out_add = 3'b001;
    #1 cs = 10'b1000000000; #0 in_add = 3'b001; #0 out_add = 3'b000;
  end

  initial
  begin
    $dumpfile("Clos16x4_test.vcd"); $dumpvars; #8 $finish;
  end
endmodule
```

Código 7 – Módulo Clos16x4Test em Verilog

O Código 8 apresenta o módulo Clos 16x4 em Verilog gerado pelo Vericonn. O módulo utiliza diversos módulos DSM de tamanho 4x4 e 4x2. No Apêndice A é apresentado

o módulo DSM. O esquemático para o módulo Clos 16x4 é apresentado na Figura 49 (gerada pela ferramenta **Xilinx Vivado 2014** (Xilinx Inc., 2014)).

```
module Clos16x4 (input cnfg, input res, input [9:0] cs,
  input [2:0] in_add, input [2:0] out_add,
  input [15:0] di, output [3:0] do
);

  wire [3:0] stg1to2_0; wire [3:0] stg1to2_1;
  wire [3:0] stg1to2_2; wire [3:0] stg1to2_3;

  DSM4x4 dsm_stg1_0 (.cnfg(cnfg), .res(res), .cs(cs[0]), .in_add(in_add
     [2:0]), .out_add(out_add[2:0]), .di(di[3:0]), .do(stg1to2_0[3:0]));
  DSM4x4 dsm_stg1_1 (.cnfg(cnfg), .res(res), .cs(cs[1]), .in_add(in_add
     [2:0]), .out_add(out_add[2:0]), .di(di[7:4]), .do(stg1to2_1[3:0]));
  DSM4x4 dsm_stg1_2 (.cnfg(cnfg), .res(res), .cs(cs[2]), .in_add(in_add
     [2:0]), .out_add(out_add[2:0]), .di(di[11:8]), .do(stg1to2_2[3:0]));
  DSM4x4 dsm_stg1_3 (.cnfg(cnfg), .res(res), .cs(cs[3]), .in_add(in_add
     [2:0]), .out_add(out_add[2:0]), .di(di[15:12]), .do(stg1to2_3[3:0]));

  wire [3:0] di_stg2_0; wire [3:0] di_stg2_1;
  wire [3:0] di_stg2_2; wire [3:0] di_stg2_3;

  assign di_stg2_0 = {stg1to2_3[0],stg1to2_2[0],stg1to2_1[0],stg1to2_0[0]};
  assign di_stg2_1 = {stg1to2_3[1],stg1to2_2[1],stg1to2_1[1],stg1to2_0[1]};
  assign di_stg2_2 = {stg1to2_3[2],stg1to2_2[2],stg1to2_1[2],stg1to2_0[2]};
  assign di_stg2_3 = {stg1to2_3[3],stg1to2_2[3],stg1to2_1[3],stg1to2_0[3]};

  wire [1:0] do_stg2_0; wire [1:0] do_stg2_1;
  wire [1:0] do_stg2_2; wire [1:0] do_stg2_3;

  DSM4x2 dsm_stg2_0 (.cnfg(cnfg), .res(res), .cs(cs[4]), .in_add(in_add
     [2:0]), .out_add(out_add[1:0]), .di(di_stg2_0[3:0]), .do(do_stg2_0
     [1:0]));
  DSM4x2 dsm_stg2_1 (.cnfg(cnfg), .res(res), .cs(cs[5]), .in_add(in_add
     [2:0]), .out_add(out_add[1:0]), .di(di_stg2_1[3:0]), .do(do_stg2_1
     [1:0]));
  DSM4x2 dsm_stg2_2 (.cnfg(cnfg), .res(res), .cs(cs[6]), .in_add(in_add
     [2:0]), .out_add(out_add[1:0]), .di(di_stg2_2[3:0]), .do(do_stg2_2
     [1:0]));
  DSM4x2 dsm_stg2_3 (.cnfg(cnfg), .res(res), .cs(cs[7]), .in_add(in_add
     [2:0]), .out_add(out_add[1:0]), .di(di_stg2_3[3:0]), .do(do_stg2_3
     [1:0]));

  wire [3:0] stg2to3_0; wire [3:0] stg2to3_1;

  assign stg2to3_0 = {do_stg2_3[0],do_stg2_2[0],do_stg2_1[0],do_stg2_0[0]};
```

```
  assign stg2to3_1 = {do_stg2_3[1],do_stg2_2[1],do_stg2_1[1],do_stg2_0[1]};

  DSM4x2 dsm_stg3_0 (.cnfg(cnfg), .res(res), .cs(cs[8]), .in_add(in_add
     [2:0]), .out_add(out_add[1:0]), .di(stg2to3_0[3:0]), .do(do[1:0]));
  DSM4x2 dsm_stg3_1 (.cnfg(cnfg), .res(res), .cs(cs[9]), .in_add(in_add
     [2:0]), .out_add(out_add[1:0]), .di(stg2to3_1[3:0]), .do(do[3:2]));

endmodule
```

Código 8 – Módulo Clos16x4 3 estágios rearranjável em Verilog

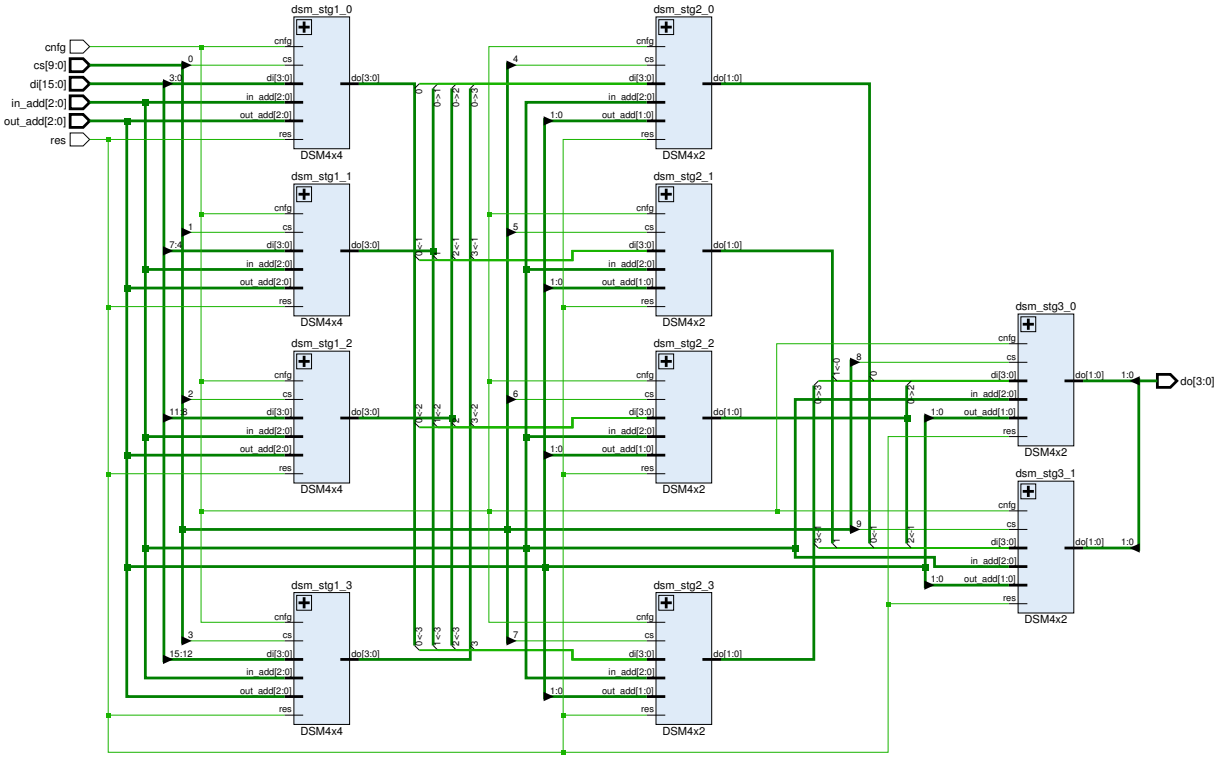


Figura 49 – Esquemático para o módulo Clos16x4

APÊNDICE C – Exemplo de Mux Tree (16x4) gerada pelo Vericonn

Nesse apêndice é apresentado um exemplo de configuração de uma rede Mux Tree 16x4 gerada pelo Vericonn. São apresentados os códigos gerados e o gráfico de onda da simulação. A arquitetura da rede Mux Tree é apresentada na seção 3.3.2.2.

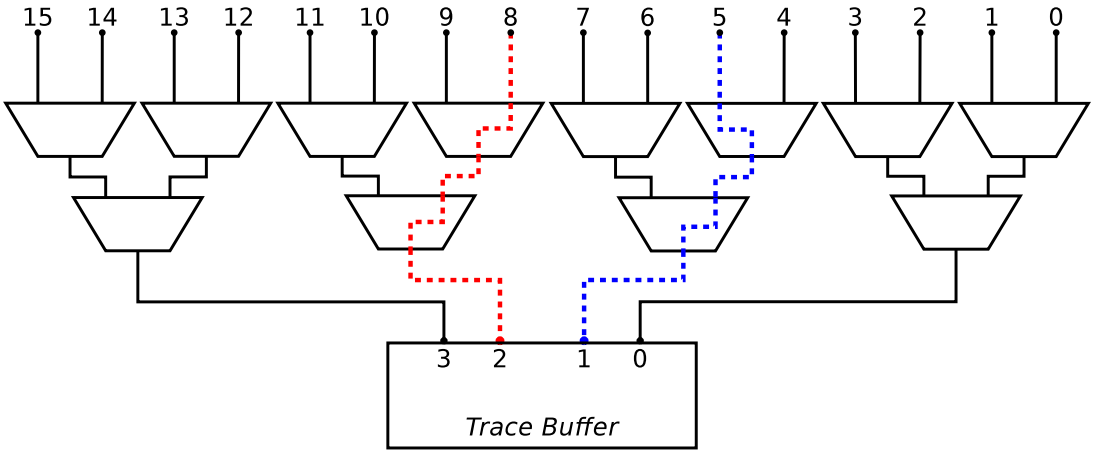


Figura 50 – Exemplo de rede Mux Tree 16x4. Nesse exemplo, uma rota liga a porta de entrada 5 à porta de saída 1. A outra rota liga a porta de entrada 8 à porta de saída 2. Para configurar cada rota são necessários 4 ciclos de *clock* para configurar cada Crossbar 2x1 individualmente.

Na Figura 50, duas rotas na rede Mux Tree 16x4 são apresentadas. Nesse exemplo, a porta de entrada 5 é conectada à porta de saída 1, e a porta de entrada 8 é conectada à porta de saída 2. Para demonstrar como criar essas rotas na arquitetura apresentada nesse trabalho, a Figura 51 mostra uma simulação para configurar a rede. Nesse exemplo, a tradução de endereços não foi habilitada no barramento **Seleciona Chip**. Logo, cada sinal do barramento **Seleciona Chip** está conectado a um Crossbar.

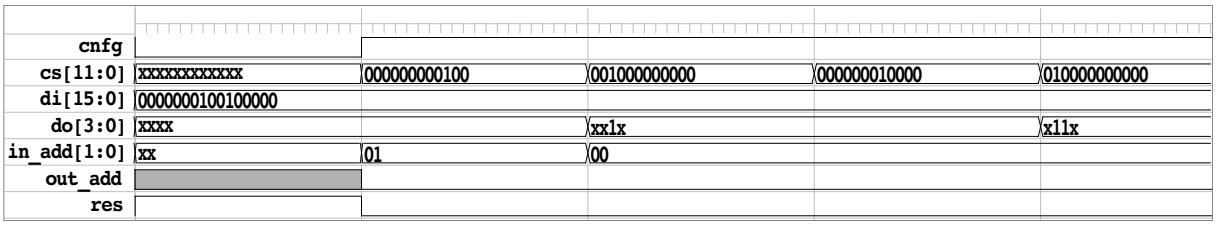


Figura 51 – Simulação de configuração de uma rede Mux Tree 16x4.

Observe que no barramento **di**, o valor das entradas nas portas que serão conectados às saídas estão em 1, o restante está em 0. No primeiro ciclo de *clock*, o valor das

entradas é colocado no barramento. Nos dois ciclos consecutivos a primeira rota é configurada e o barramento **do** apresenta o valor 1 no bit 1. Nos dois ciclos seguintes, a segunda rota é configurada e outro 1 pode ser visto no bit 2 do barramento **do**. O código dessa simulação está exposto no Código 9.

```
module Mux_Tree16x4_test;
  reg cnfg; reg res; reg [11:0] cs;
  reg [1:0] in_add; reg out_add;
  reg [15:0] di;  wire [3:0] do;

  Mux_Tree16x4 mux_tree(.cnfg(cnfg), .res(res), .cs(cs),
  .in_add(in_add), .out_add(out_add), .di(di), .do(do));

  initial
  begin
    #0 res = 1; #0 cnfg = 0; #0 di = 16'b0000000100100000;
    #1 res = 0; #0 cnfg = 1; #0 out_add = 0;
    // rota azul
    #0 cs = 12'b000000000100; #0 in_add = 2'b01;
    #1 cs = 12'b001000000000; #0 in_add = 2'b00;
    // rota vermelha
    #1 cs = 12'b000000010000; #0 in_add = 2'b00;
    #1 cs = 12'b010000000000; #0 in_add = 2'b00;
  end

  initial
  begin
    $dumpfile("Mux_Tree16x4_test.vcd"); $dumpvars; #5 $finish;
  end
endmodule
```

Código 9 – Módulo MuxTree16x4Test em Verilog

```
module Mux_Tree16x4 (input cnfg, input res, input [11:0] cs,
  input [1:0] in_add, input out_add, input [15:0] di,output [3:0] do
);
  wire s0_to_s1_0;  assign s0_to_s1_0 = di[0];
  wire s0_to_s1_1;  assign s0_to_s1_1 = di[1];
  wire s0_to_s1_2;  assign s0_to_s1_2 = di[2];
  wire s0_to_s1_3;  assign s0_to_s1_3 = di[3];
  wire s0_to_s1_4;  assign s0_to_s1_4 = di[4];
  wire s0_to_s1_5;  assign s0_to_s1_5 = di[5];
  wire s0_to_s1_6;  assign s0_to_s1_6 = di[6];
  wire s0_to_s1_7;  assign s0_to_s1_7 = di[7];
  wire s0_to_s1_8;  assign s0_to_s1_8 = di[8];
```

```
  wire s0_to_s1_9;  assign s0_to_s1_9 = di[9];
  wire s0_to_s1_10; assign s0_to_s1_10 = di[10];
  wire s0_to_s1_11; assign s0_to_s1_11 = di[11];
  wire s0_to_s1_12; assign s0_to_s1_12 = di[12];
  wire s0_to_s1_13; assign s0_to_s1_13 = di[13];
  wire s0_to_s1_14; assign s0_to_s1_14 = di[14];
  wire s0_to_s1_15; assign s0_to_s1_15 = di[15];

  wire s1_to_s2_0; wire s1_to_s2_1; wire s1_to_s2_2; wire s1_to_s2_3;
  wire s1_to_s2_4; wire s1_to_s2_5; wire s1_to_s2_6; wire s1_to_s2_7;

  DSM2x1 dsm_s1_0 (.cnfg(cnfg), .res(res), .cs(cs[0]), .in_add(in_add[1:0])
     , .out_add(out_add), .di({s0_to_s1_1,s0_to_s1_0}), .do(s1_to_s2_0));
  DSM2x1 dsm_s1_1 (.cnfg(cnfg), .res(res), .cs(cs[1]), .in_add(in_add[1:0])
     , .out_add(out_add), .di({s0_to_s1_3,s0_to_s1_2}), .do(s1_to_s2_1));
  DSM2x1 dsm_s1_2 (.cnfg(cnfg), .res(res), .cs(cs[2]), .in_add(in_add[1:0])
     , .out_add(out_add), .di({s0_to_s1_5,s0_to_s1_4}), .do(s1_to_s2_2));
  DSM2x1 dsm_s1_3 (.cnfg(cnfg), .res(res), .cs(cs[3]), .in_add(in_add[1:0])
     , .out_add(out_add), .di({s0_to_s1_7,s0_to_s1_6}), .do(s1_to_s2_3));
  DSM2x1 dsm_s1_4 (.cnfg(cnfg), .res(res), .cs(cs[4]), .in_add(in_add[1:0])
     , .out_add(out_add), .di({s0_to_s1_9,s0_to_s1_8}), .do(s1_to_s2_4));
  DSM2x1 dsm_s1_5 (.cnfg(cnfg), .res(res), .cs(cs[5]), .in_add(in_add[1:0])
     , .out_add(out_add), .di({s0_to_s1_11,s0_to_s1_10}), .do(s1_to_s2_5));
  DSM2x1 dsm_s1_6 (.cnfg(cnfg), .res(res), .cs(cs[6]), .in_add(in_add[1:0])
     , .out_add(out_add), .di({s0_to_s1_13,s0_to_s1_12}), .do(s1_to_s2_6));
  DSM2x1 dsm_s1_7 (.cnfg(cnfg), .res(res), .cs(cs[7]), .in_add(in_add[1:0])
     , .out_add(out_add), .di({s0_to_s1_15,s0_to_s1_14}), .do(s1_to_s2_7));

  wire s2_to_s3_0; wire s2_to_s3_1; wire s2_to_s3_2; wire s2_to_s3_3;

  DSM2x1 dsm_s2_0 (.cnfg(cnfg), .res(res), .cs(cs[8]), .in_add(in_add[1:0])
     , .out_add(out_add), .di({s1_to_s2_1,s1_to_s2_0}), .do(s2_to_s3_0));
  DSM2x1 dsm_s2_1 (.cnfg(cnfg), .res(res), .cs(cs[9]), .in_add(in_add[1:0])
     , .out_add(out_add), .di({s1_to_s2_3,s1_to_s2_2}), .do(s2_to_s3_1));
  DSM2x1 dsm_s2_2 (.cnfg(cnfg), .res(res), .cs(cs[10]), .in_add(in_add
     [1:0]), .out_add(out_add), .di({s1_to_s2_5,s1_to_s2_4}), .do(
     s2_to_s3_2));
  DSM2x1 dsm_s2_3 (.cnfg(cnfg), .res(res), .cs(cs[11]), .in_add(in_add
     [1:0]), .out_add(out_add), .di({s1_to_s2_7,s1_to_s2_6}), .do(
     s2_to_s3_3));

  assign do[0] = s2_to_s3_0; assign do[1] = s2_to_s3_1;
  assign do[2] = s2_to_s3_2; assign do[3] = s2_to_s3_3;
endmodule
```

Código 10 – Módulo MuxTree16x4 em Verilog

O Código 10 apresenta o módulo em Verilog gerado pelo Vericonn. O módulo utiliza diversos módulos DSM de tamanho 2x1. No Apêndice A é apresentado o módulo DSM. O esquemático para o módulo Mux Tree 16x4 é apresentado na Figura 52 (gerada pela ferramenta **Xilinx Vivado 2014** (Xilinx Inc., 2014)).

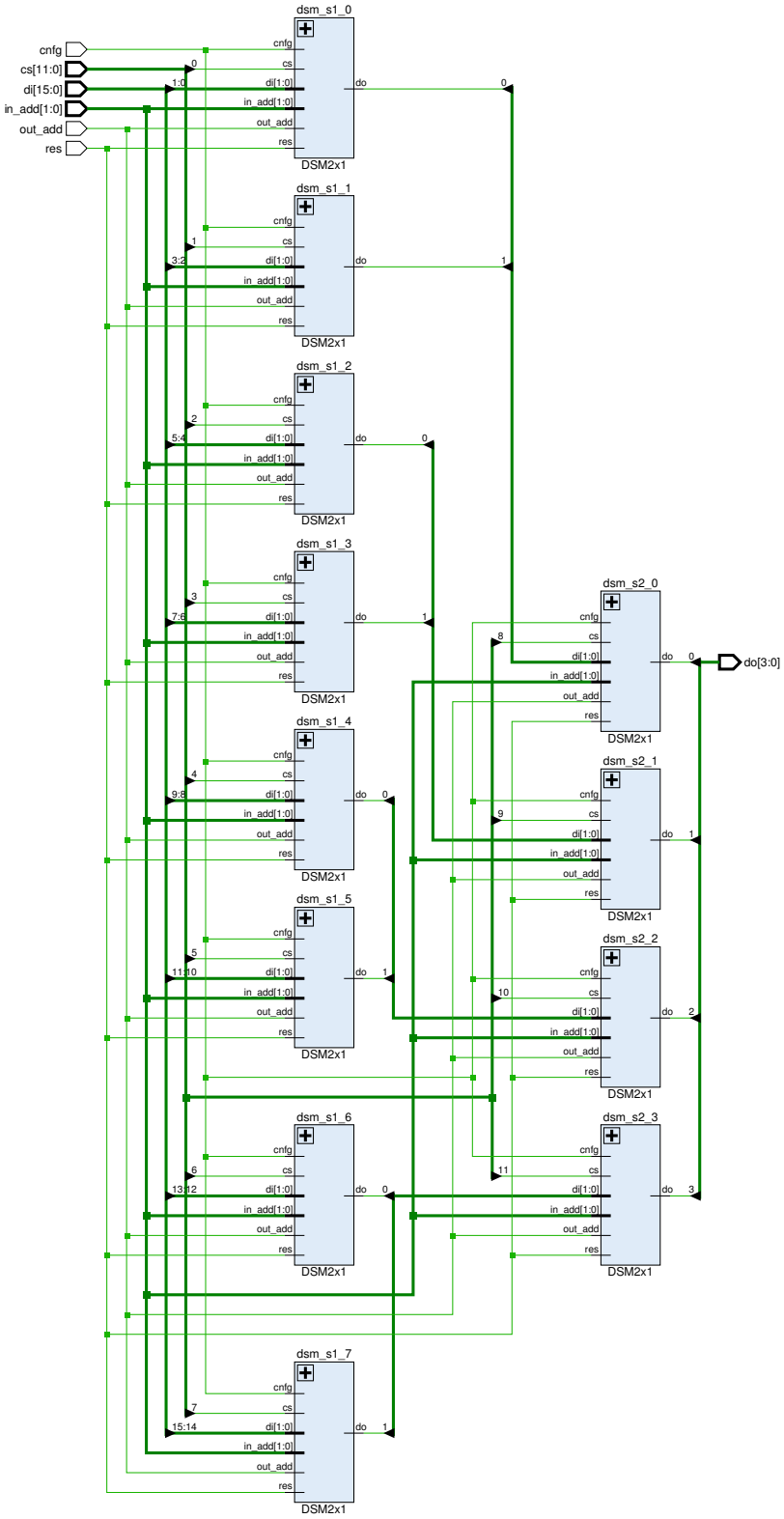


Figura 52 – Esquemático para o módulo MuxTree16x4

APÊNDICE D – Exemplo de Ômega (16x4) gerada pelo Vericonn

Nesse apêndice é apresentado um exemplo de configuração de uma rede Ômega 16x4 gerada pelo Vericonn. São apresentados os códigos gerados e o gráfico de onda da simulação. A arquitetura da rede Ômega é apresentada na seção 3.3.2.3.

Na Figura 53 duas rotas são apresentadas. A primeira rota conecta a porta de entrada 0000_b na porta de saída 0011_b. Na segunda rota, a porta de entrada 0101_b é conectada à porta de saída 0000_b. Para configurar essas rotas, é necessário configurar individualmente cada um dos 8 Crossbares.

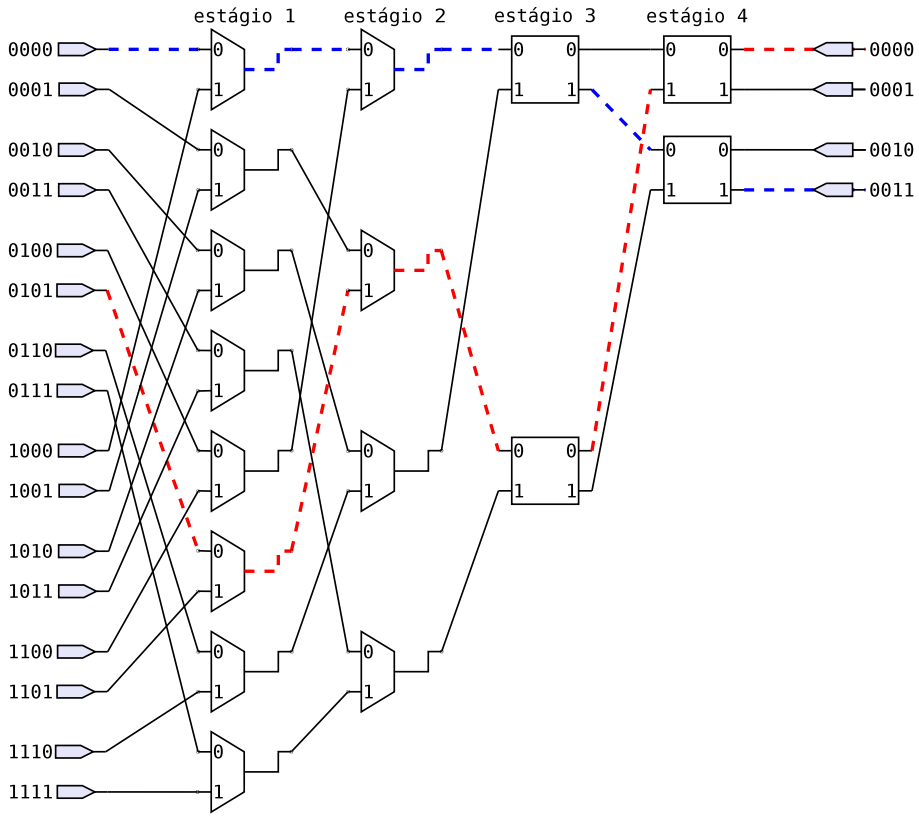


Figura 53 – Exemplo de rede Ômega 16x4. Nesse exemplo, uma rota liga a porta de entrada 0 a porta de saída 3. A outra rota liga a porta de entrada 5 a porta de saída 0. Para configurar essas rotas deve-se configurar cada Crossbar individualmente, com o tempo de 8 ciclos de *clock*.

Na Figura 54 é apresentado o gráfico de ondas para a simulação da rede apresentada na Figura 53. O código da simulação em LDH Verilog é apresentado no Código 11. A simulação apresenta 9 ciclos de *clock*. O primeiro ciclo é apenas um *reset* na rede. Os 4

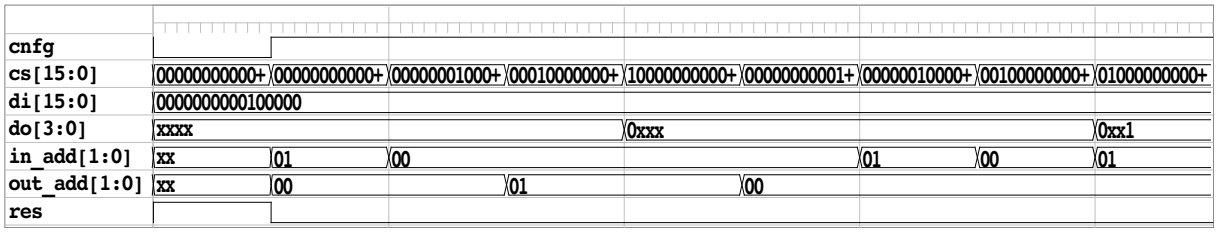


Figura 54 – Simulação de configuração de uma rede Ômega 16x4.

ciclos seguintes configuram a rota azul. O bit na posição 0 do barramento **di** tem o valor 0, logo, no quinto ciclo pode ser observado que a saída da rede apresenta o valor 0 na porta de saída 3. O bit na posição 5 do barramento **di** tem o valor 1, e no último ciclo pode ser observado esse valor na saída 0 da rede.

```
module Omega16x4_test;
  reg cnfg; reg res; reg [15:0] cs;
  reg [1:0] in_add; reg [1:0] out_add;
  reg [15:0] di; wire [3:0] do;

  Omega16x4 omega(.cnfg(cnfg), .res(res), .cs(cs),
  .in_add(in_add), .out_add(out_add), .di(di), .do(do));

  initial
  begin
    #0 cs = 16'b0000000000000000; #0 di = 16'b0000000000100000;
    #0 res = 1;  #0 cnfg = 0; #1 cnfg = 1; #0 res = 0;
    // rota azul
    #0 cs = 16'b0000000000000001; #0 in_add = 1; #0 out_add = 0;
    #1 cs = 16'b0000000100000000; #0 in_add = 0; #0 out_add = 0;
    #1 cs = 16'b0001000000000000; #0 in_add = 0; #0 out_add = 1;
    #1 cs = 16'b1000000000000000; #0 in_add = 0; #0 out_add = 1;
    // rota vermelha
    #1 cs = 16'b0000000000100000; #0 in_add = 0; #0 out_add = 0;
    #1 cs = 16'b0000001000000000; #0 in_add = 1; #0 out_add = 0;
    #1 cs = 16'b0010000000000000; #0 in_add = 0; #0 out_add = 0;
    #1 cs = 16'b0100000000000000; #0 in_add = 1; #0 out_add = 0;
  end

  initial
  begin
    $dumpfile("Omega16x4_test.vcd"); $dumpvars; #9 $finish;
  end
endmodule
```

Código 11 – Módulo Omega16x4Test em Verilog

O Código 12 apresenta o módulo em Verilog gerado pelo Vericonn. O módulo utiliza diversos módulos DSM de tamanho 2x1 e 2x2. No Apêndice A é apresentado o módulo DSM. O esquemático para o módulo Ômega 16x4 é apresentado na Figura 55 (gerada pela ferramenta **Xilinx Vivado 2014** (Xilinx Inc., 2014)).

```
module Omega16x4 (input cnfg, input res, input [15:0] cs,
  input [1:0] in_add, input [1:0] out_add, input [15:0] di, output [3:0] do
);
  wire s0_to_s1_0;  wire s0_to_s1_1;  wire s0_to_s1_2;  wire s0_to_s1_3;
  wire s0_to_s1_4;  wire s0_to_s1_5;  wire s0_to_s1_6;  wire s0_to_s1_7;
  wire s0_to_s1_8;  wire s0_to_s1_9;  wire s0_to_s1_10; wire s0_to_s1_11;
  wire s0_to_s1_12; wire s0_to_s1_13; wire s0_to_s1_14; wire s0_to_s1_15;
  assign s0_to_s1_0 = di[0]; assign s0_to_s1_1 = di[1];
  assign s0_to_s1_2 = di[2]; assign s0_to_s1_3 = di[3];
  assign s0_to_s1_4 = di[4]; assign s0_to_s1_5 = di[5];
  assign s0_to_s1_6 = di[6]; assign s0_to_s1_7 = di[7];
  assign s0_to_s1_8 = di[8]; assign s0_to_s1_9 = di[9];
  assign s0_to_s1_10 = di[10]; assign s0_to_s1_11 = di[11];
  assign s0_to_s1_12 = di[12]; assign s0_to_s1_13 = di[13];
  assign s0_to_s1_14 = di[14]; assign s0_to_s1_15 = di[15];

  wire s1_to_s2_0;  wire s1_to_s2_2;  wire s1_to_s2_4;  wire s1_to_s2_6;
  wire s1_to_s2_8;  wire s1_to_s2_10; wire s1_to_s2_12; wire s1_to_s2_14;
  DSM2x1 dsm_s1_0 (.cnfg(cnfg), .res(res), .cs(cs[0]), .in_add(in_add[1:0])
     , .out_add(out_add[0]), .di({s0_to_s1_8,s0_to_s1_0}), .do(s1_to_s2_0))
     ;
  DSM2x1 dsm_s1_1 (.cnfg(cnfg), .res(res), .cs(cs[1]), .in_add(in_add[1:0])
     , .out_add(out_add[0]), .di({s0_to_s1_9,s0_to_s1_1}), .do(s1_to_s2_2))
     ;
  DSM2x1 dsm_s1_2 (.cnfg(cnfg), .res(res), .cs(cs[2]), .in_add(in_add[1:0])
     , .out_add(out_add[0]), .di({s0_to_s1_10,s0_to_s1_2}), .do(s1_to_s2_4)
     );
  DSM2x1 dsm_s1_3 (.cnfg(cnfg), .res(res), .cs(cs[3]), .in_add(in_add[1:0])
     , .out_add(out_add[0]), .di({s0_to_s1_11,s0_to_s1_3}), .do(s1_to_s2_6)
     );
  DSM2x1 dsm_s1_4 (.cnfg(cnfg), .res(res), .cs(cs[4]), .in_add(in_add[1:0])
     , .out_add(out_add[0]), .di({s0_to_s1_12,s0_to_s1_4}), .do(s1_to_s2_8)
     );
  DSM2x1 dsm_s1_5 (.cnfg(cnfg), .res(res), .cs(cs[5]), .in_add(in_add[1:0])
     , .out_add(out_add[0]), .di({s0_to_s1_13,s0_to_s1_5}), .do(s1_to_s2_10
     ));
  DSM2x1 dsm_s1_6 (.cnfg(cnfg), .res(res), .cs(cs[6]), .in_add(in_add[1:0])
     , .out_add(out_add[0]), .di({s0_to_s1_14,s0_to_s1_6}), .do(s1_to_s2_12
     ));
  DSM2x1 dsm_s1_7 (.cnfg(cnfg), .res(res), .cs(cs[7]), .in_add(in_add[1:0])
     , .out_add(out_add[0]), .di({s0_to_s1_15,s0_to_s1_7}), .do(s1_to_s2_14
     ));
```

```
  wire s2_to_s3_0; wire s2_to_s3_4; wire s2_to_s3_8; wire s2_to_s3_12;
  DSM2x1 dsm_s2_0 (.cnfg(cnfg), .res(res), .cs(cs[8]), .in_add(in_add[1:0])
     , .out_add(out_add[0]), .di({s1_to_s2_8,s1_to_s2_0}), .do(s2_to_s3_0))
     ;
  DSM2x1 dsm_s2_2 (.cnfg(cnfg), .res(res), .cs(cs[9]), .in_add(in_add[1:0])
     , .out_add(out_add[0]), .di({s1_to_s2_10,s1_to_s2_2}), .do(s2_to_s3_4)
     );
  DSM2x1 dsm_s2_4 (.cnfg(cnfg), .res(res), .cs(cs[10]), .in_add(in_add
     [1:0]), .out_add(out_add[0]), .di({s1_to_s2_12,s1_to_s2_4}), .do(
     s2_to_s3_8));
  DSM2x1 dsm_s2_6 (.cnfg(cnfg), .res(res), .cs(cs[11]), .in_add(in_add
     [1:0]), .out_add(out_add[0]), .di({s1_to_s2_14,s1_to_s2_6}), .do(
     s2_to_s3_12));

  wire s3_to_s4_0; wire s3_to_s4_1; wire s3_to_s4_8; wire s3_to_s4_9;
  DSM2x2 dsm_s3_0 (.cnfg(cnfg), .res(res), .cs(cs[12]), .in_add(in_add
     [1:0]), .out_add(out_add[1:0]), .di({s2_to_s3_8,s2_to_s3_0}), .do({
     s3_to_s4_1,s3_to_s4_0}));
  DSM2x2 dsm_s3_4 (.cnfg(cnfg), .res(res), .cs(cs[13]), .in_add(in_add
     [1:0]), .out_add(out_add[1:0]), .di({s2_to_s3_12,s2_to_s3_4}), .do({
     s3_to_s4_9,s3_to_s4_8}));

  wire s4_to_s5_0; wire s4_to_s5_1; wire s4_to_s5_2; wire s4_to_s5_3;
  DSM2x2 dsm_s4_0 (.cnfg(cnfg), .res(res), .cs(cs[14]), .in_add(in_add
     [1:0]), .out_add(out_add[1:0]), .di({s3_to_s4_8,s3_to_s4_0}), .do({
     s4_to_s5_1,s4_to_s5_0}));
  DSM2x2 dsm_s4_1 (.cnfg(cnfg), .res(res), .cs(cs[15]), .in_add(in_add
     [1:0]), .out_add(out_add[1:0]), .di({s3_to_s4_9,s3_to_s4_1}), .do({
     s4_to_s5_3,s4_to_s5_2}));

  assign do[0] = s4_to_s5_0; assign do[1] = s4_to_s5_1;
  assign do[2] = s4_to_s5_2; assign do[3] = s4_to_s5_3;

endmodule
```

Código 12 – Módulo Omega16x4 em Verilog

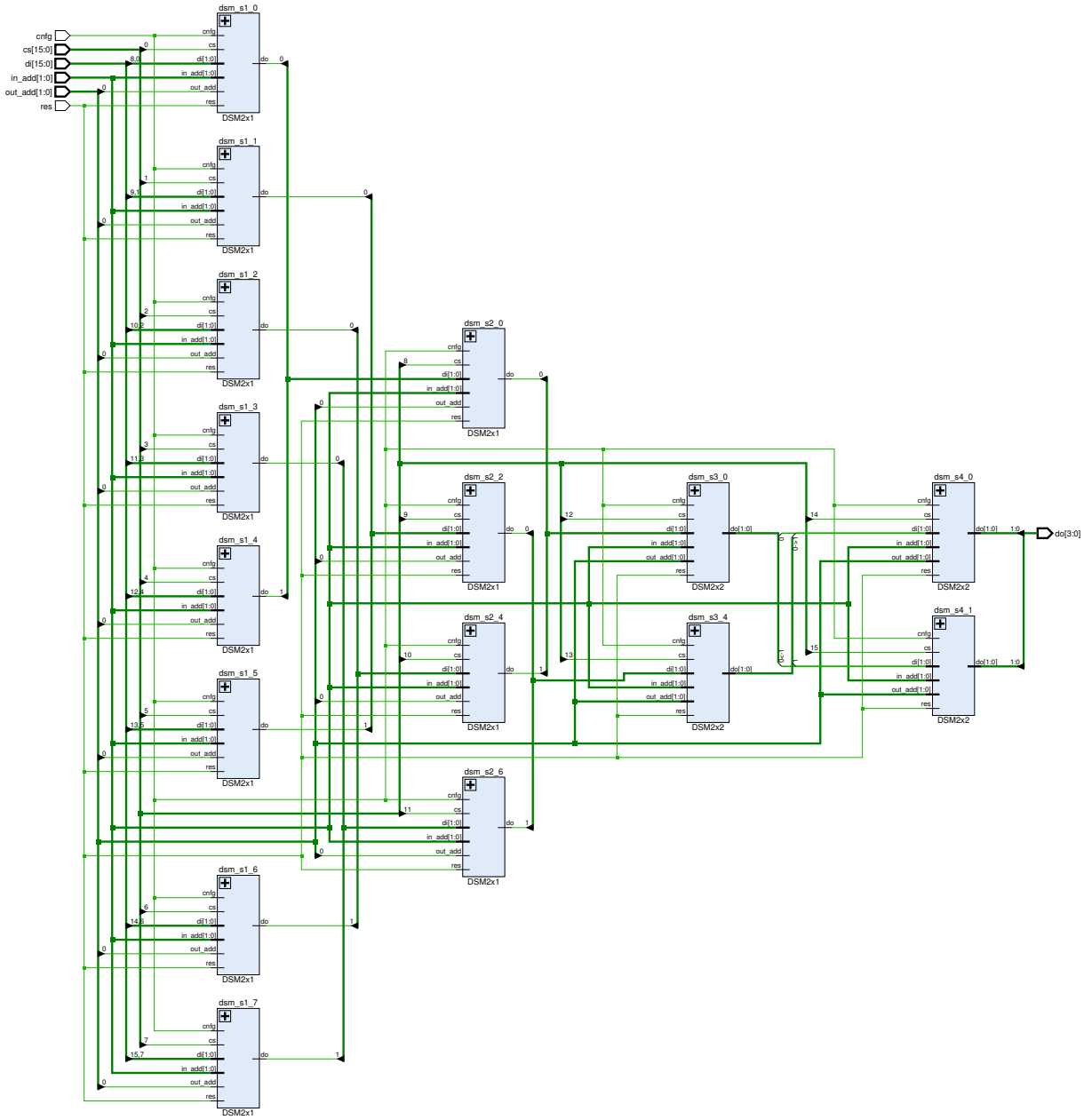
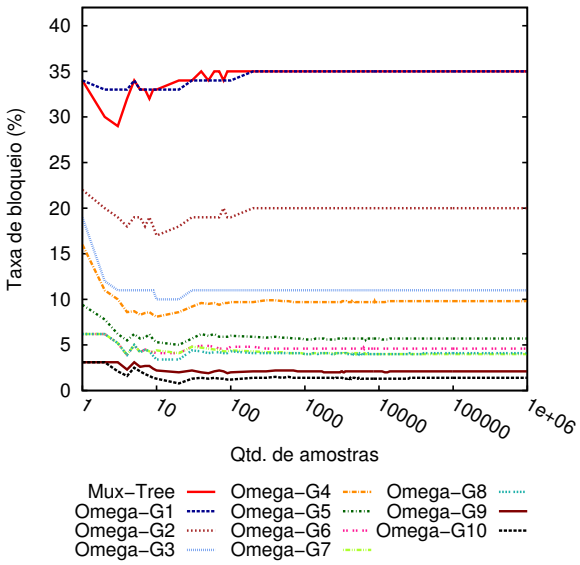


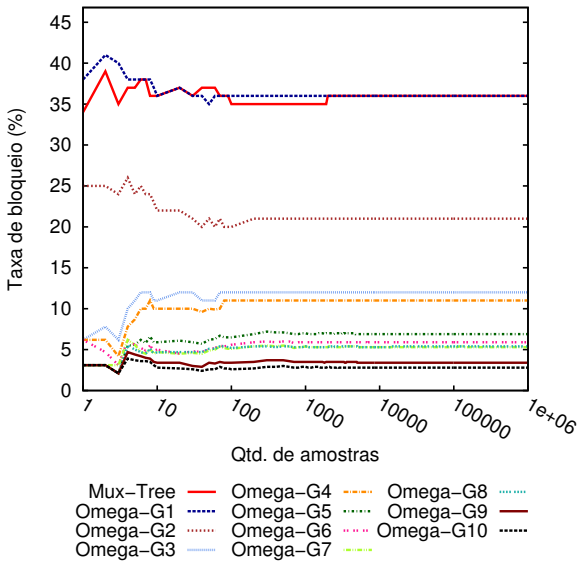
Figura 55 – Esquemático para o módulo Omega16x4

APÊNDICE E – Quantidade de amostras versus taxa de bloqueio

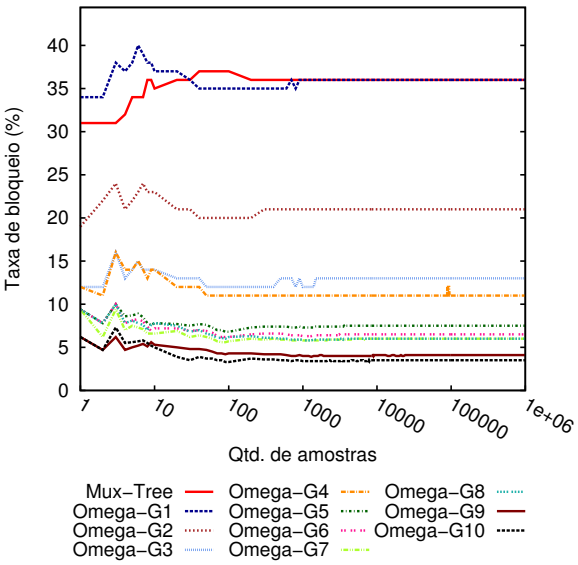
Conforme é apresentado na seção 4.1, nesse trabalho foram utilizadas 1 milhão de amostras para calcular as taxa de bloqueio para as redes Ômega e Mux Tree. Nesse apêndice é apresentado um comparativo entre a quantidade de amostras versus a taxa de bloqueio. Na Figura 56 cada gráfico apresenta no eixo y a taxa de bloqueio para um determinado tamanho de rede. No eixo x é utilizada uma escala logarítmica na base 10 para apresentar a quantidade de amostras. É possível notar que a taxa de bloqueio apresenta bastante variação até ≈ 100 amostras. Até ≈ 10.000 amostras existe pouca variação na taxa de bloqueio, e após ≈ 100.000 amostras a taxa de bloqueio converge.



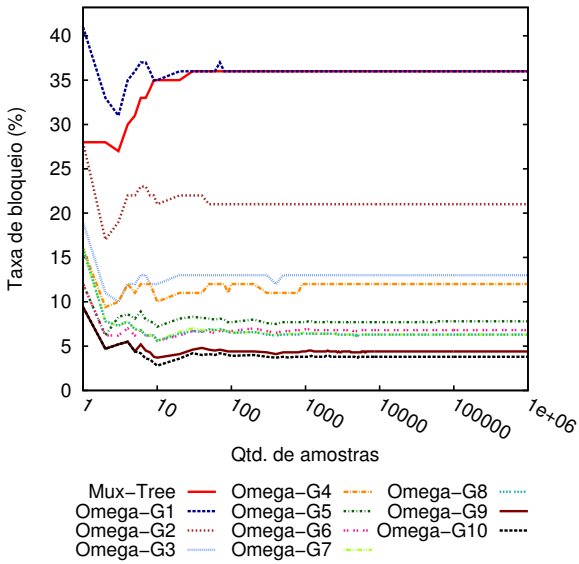
(a) 512x32



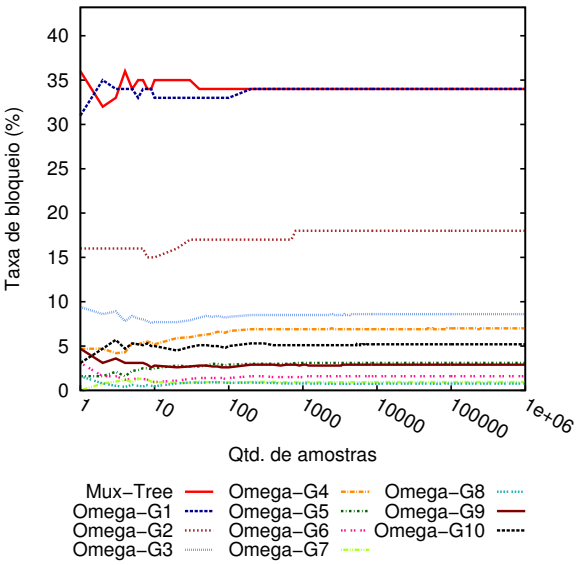
(b) 1024x32



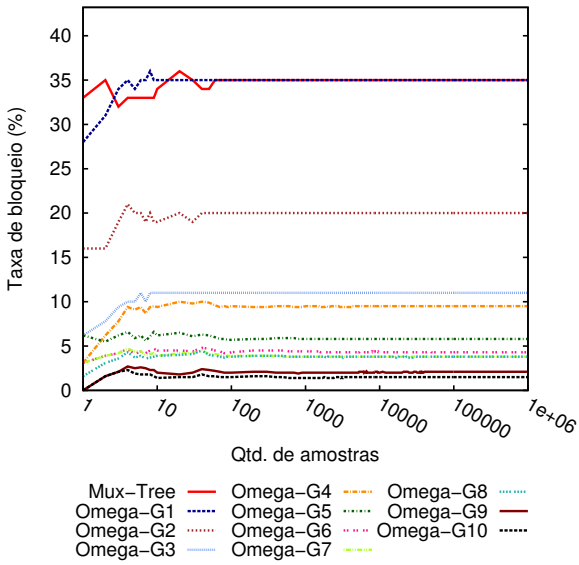
(c) 2048x32



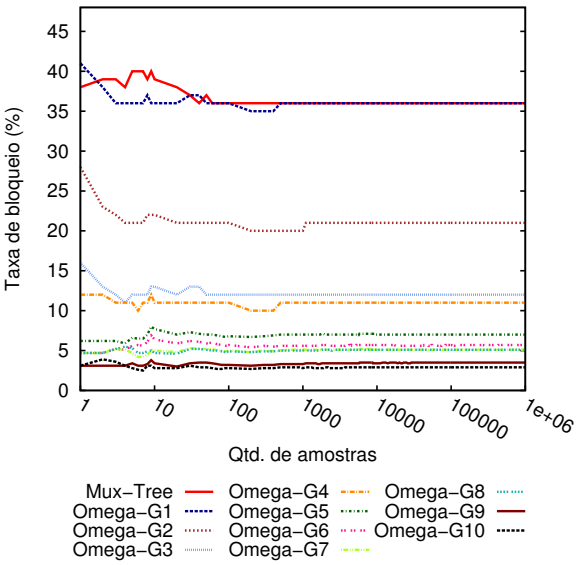
(d) 4096x32



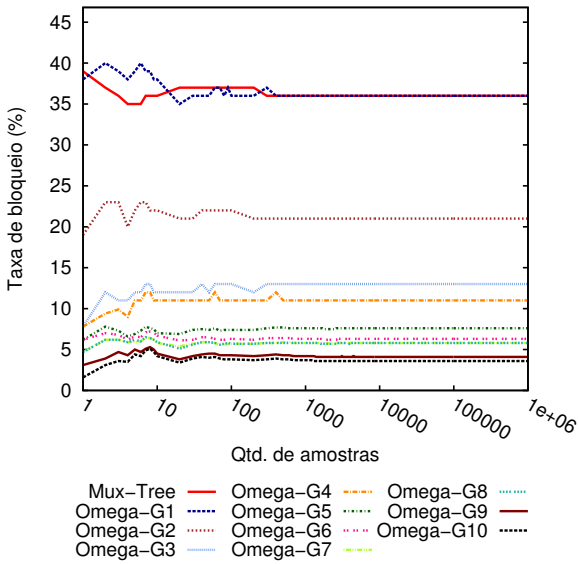
(e) 512x64



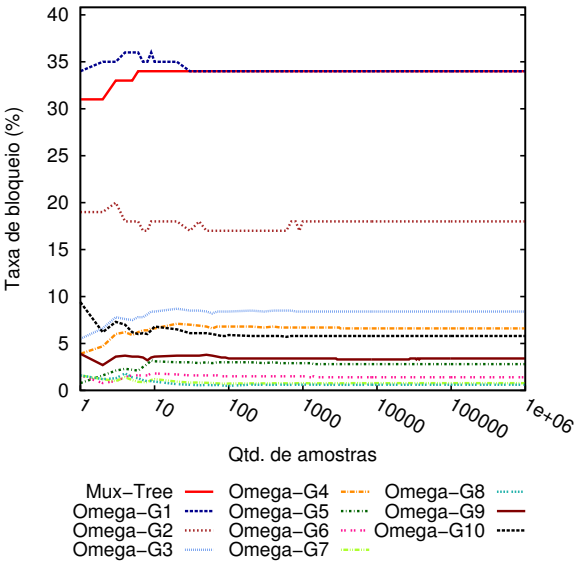
(f) 1024x64



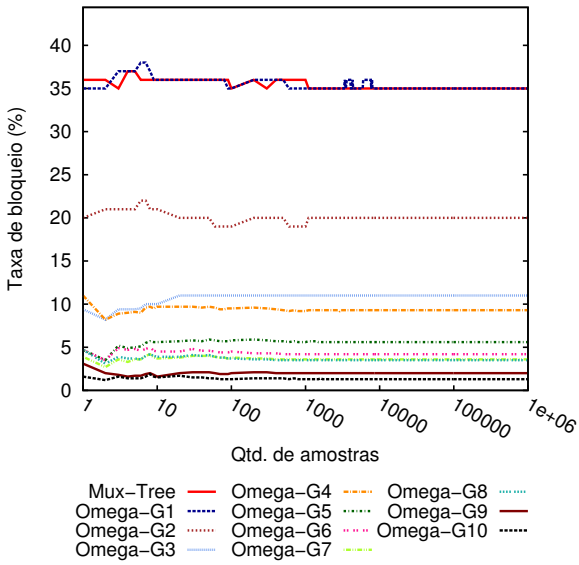
(g) 2048x64



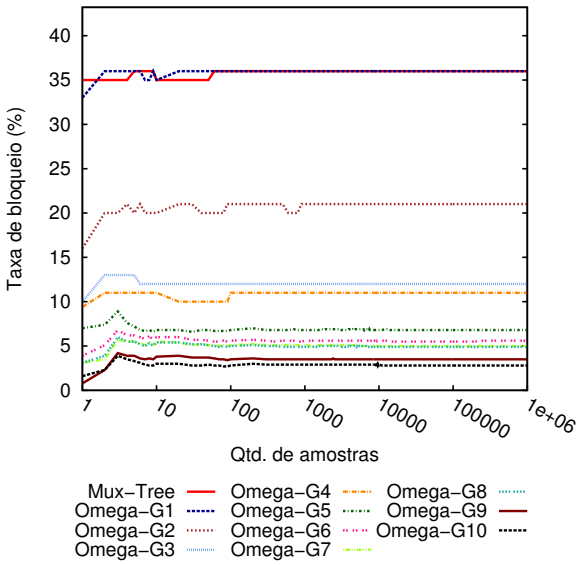
(h) 4096x64



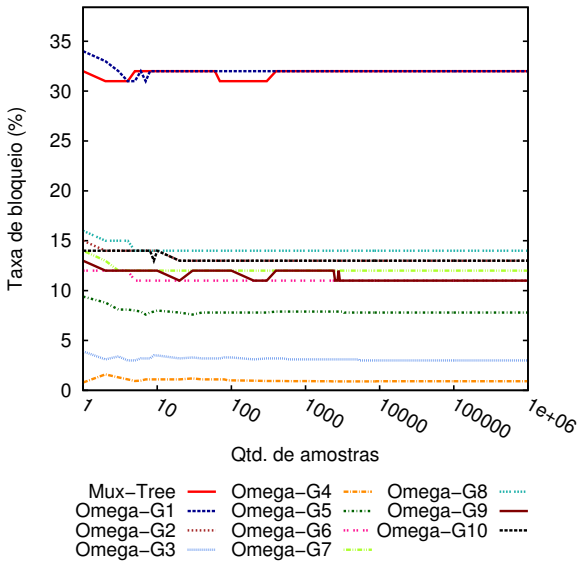
(i) 1024x128



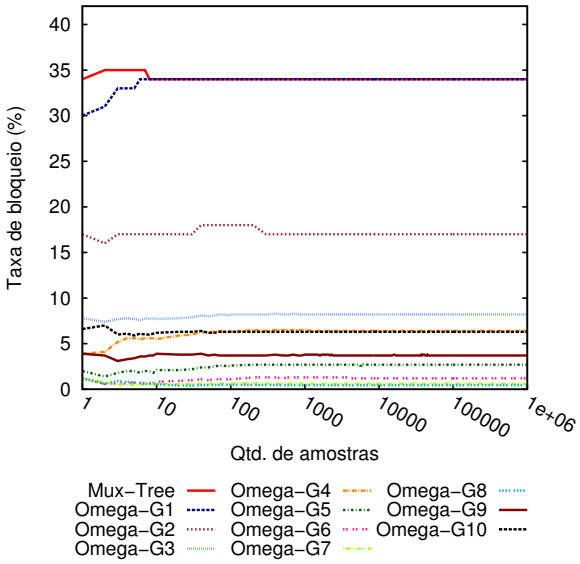
(j) 2048x128



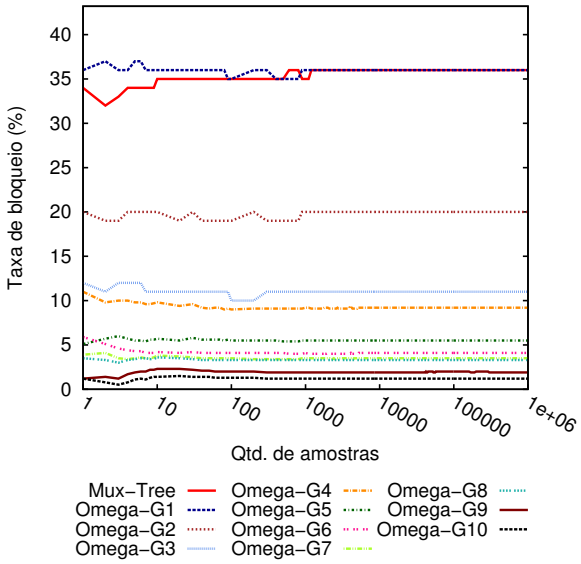
(k) 4096x128



(l) 1024x256



(m) 2048x256



(n) 4096x256

Figura 56 – Comparação da quantidade de amostras e a taxa de bloqueio para várias redes Mux Tree e Omega.

APÊNDICE F – Detalhamento dos resultados

Nas tabelas 1–14 são apresentados resultados obtidos nesse trabalho. Cada linha apresenta os dados de uma determinada rede. Para apresentar o resultado das diversas redes Ômega, e conforme é demonstrado na seção 3, a letra **G** seguida por um número indica a disposição das saídas disponíveis da rede. Para redes Clos, **R** seguido de um número representa que a rede é **rearranjável** e a quantidade de estágios da rede.

A coluna **Muxes** apresenta a quantidade de multiplexadores 2x1 utilizados na rede. Essa métrica é apresentada apenas para as rede que utilizam unicamente esse tamanho de multiplexador, nesse experimento: Mux Tree e Ômega. A coluna δm apresenta a quantidade de multiplexadores utilizados pela rede i dividido pela quantidade de multiplexadores utilizados pela Mux Tree, sendo i uma rede qualquer. A coluna **Gates** apresenta o tamanho da área em termos de equivalência de portas NAND. δg apresenta a quantidade de **Gates utilizados pela rede** i dividido pela quantidade de **Gates da rede Mux Tree**.

Os valores de área apresentados nesse trabalho levam em consideração a área total do circuito com portas e fios. Esse valor é calculado pela ferramenta de síntese **Encounter RTL Compiler 11.25** (Cadence Design Systems, 2015). Para obter o valor em NANDs equivalentes, a área total é dividida pelo tamanho da maior célula NAND da biblioteca de células utilizada na síntese. Dessa forma, foi utilizado a célula NAND2 cujo tamanho é $8.96\mu m * 6.72\mu m = 60.211\mu m$ (IBM Systems and Technology Group, 2010, p. 48).

Nas colunas **%Blq**, **%Min**, **%Max** e **%Amtr** são apresentadas as taxas de bloqueio (em porcentagem) calculadas nos experimentos. Os dados foram obtidos através de experimentos com 1 milhão de amostras[1]. A coluna **%Blk** apresenta a taxa média de bloqueio da rede. A coluna **%Min** apresenta a menor taxa de bloqueio obtido nas amostras, e a coluna **%Max** apresenta a maior taxa de bloqueio. A coluna **%Amtr** apresenta a taxa de amostras que contêm bloqueio. Para exemplificar, na Tabela 1, a rede Mux Tree não obteve nenhuma amostra sem bloqueio, e a rede Ômega G2 obteve cerca de 100 amostras com nenhum bloqueio.

[1] Para as redes Crossbar e Clos que são redes não bloqueantes, a tabela foi preenchida com zeros nessas colunas.

Tabela 1 – Comparação dos resultados para as redes Mux Tree, Ômega, Crossbar e Clos de tamanho 512x32.

Rede	**Muxes**	δm	**Gates**	δg	**%Blq**	**%Min**	**%Max**	**%Amtr**
Mux Tree	480	1.00	2154.622	1.00	35.05	9.37	62.50	100.00
Ômega G1	640	1.33	2810.117	1.30	35.05	9.37	59.37	100.00
Ômega G2	800	1.66	3465.512	1.61	19.72	0.00	46.87	99.99
Ômega G3	992	2.06	4252.013	1.97	11.21	0.00	37.50	99.52
Ômega G4	1120	2.33	4776.336	2.22	9.77	0.00	34.37	98.59
Ômega G5	1408	2.93	5956.071	2.76	5.70	0.00	28.12	89.76
Ômega G6	1504	3.13	6349.304	2.95	4.63	0.00	25.00	84.08
Ômega G7	1664	3.46	7004.716	3.25	4.03	0.00	21.87	79.11
Ômega G8	1760	3.66	7397.967	3.43	4.05	0.00	21.87	78.31
Ômega G9	2272	4.73	9495.274	4.41	2.06	0.00	21.87	50.54
Ômega G10	2336	4.86	9757.436	4.53	1.35	0.00	18.75	36.67
Crossbar	-	-	10143.977	4.71	0.00	0.00	0.00	0.00
Clos R3	-	-	19812.459	9.20	0.00	0.00	0.00	0.00
Clos R5	-	-	15963.727	7.41	0.00	0.00	0.00	0.00
Clos R7	-	-	20294.896	9.42	0.00	0.00	0.00	0.00

Tabela 2 – Comparação dos resultados para as redes Mux Tree, Ômega, Crossbar e Clos de tamanho 1024x32.

Rede	**Muxes**	δm	**Gates**	δg	**%Blq**	**%Min**	**%Max**	**%Amtr**
Mux Tree	992	1.00	4447.061	1.00	35.62	9.37	62.50	100.00
Ômega G1	1152	1.16	5102.556	1.15	35.62	12.50	62.50	100.00
Ômega G2	1312	1.32	5757.951	1.29	20.63	0.00	46.87	99.99
Ômega G3	1504	1.51	6544.452	1.47	12.27	0.00	37.50	99.71
Ômega G4	1632	1.64	7068.774	1.59	10.92	0.00	37.50	99.13
Ômega G5	1920	1.93	8248.509	1.85	6.89	0.00	31.25	93.80
Ômega G6	2016	2.03	8641.759	1.94	5.90	0.00	28.12	90.50
Ômega G7	2176	2.19	9297.155	2.09	5.34	0.00	28.12	87.56
Ômega G8	2272	2.29	9690.405	2.18	5.37	0.00	28.12	87.06
Ômega G9	2784	2.80	11787.713	2.65	3.43	0.00	25.00	71.03
Ômega G10	2848	2.87	12049.874	2.71	2.80	0.00	21.87	63.44
Crossbar	-	-	20814.303	4.68	0.00	0.00	0.00	0.00
Clos R3	-	-	50768.165	11.42	0.00	0.00	0.00	0.00
Clos R5	-	-	41165.584	9.26	0.00	0.00	0.00	0.00
Clos R7	-	-	41574.878	9.35	0.00	0.00	0.00	0.00

Tabela 3 – Comparação dos resultados para as redes Mux Tree, Ômega, Crossbar e Clos de tamanho 2048x32.

Rede	**Muxes**	δm	**Gates**	δg	**%Blq**	**%Min**	**%Max**	**%Amtr**
Mux Tree	2016	1.00	9031.937	1.00	35.93	6.25	62.50	100.00
Ômega G1	2176	1.07	9687.432	1.07	35.91	12.50	62.50	100.00
Ômega G2	2336	1.15	10342.844	1.15	21.06	0.00	53.12	99.99
Ômega G3	2528	1.25	11129.328	1.23	12.78	0.00	37.50	99.77
Ômega G4	2656	1.31	11653.651	1.29	11.47	0.00	37.50	99.33
Ômega G5	2944	1.46	12833.385	1.42	7.47	0.00	31.25	95.20
Ômega G6	3040	1.50	13226.636	1.46	6.51	0.00	28.12	92.58
Ômega G7	3200	1.58	13882.031	1.54	5.97	0.00	28.12	90.31
Ômega G8	3296	1.63	14275.281	1.58	6.00	0.00	28.12	89.88
Ômega G9	3808	1.88	16372.589	1.81	4.06	0.00	25.00	77.37
Ômega G10	3872	1.92	16634.751	1.84	3.47	0.00	21.87	71.58
Crossbar	-	-	40582.783	4.49	0.00	0.00	0.00	0.00
Clos R3	-	-	77684.542	8.60	0.00	0.00	0.00	0.00
Clos R5	-	-	87288.502	9.66	0.00	0.00	0.00	0.00
Clos R7	-	-	85965.421	9.52	0.00	0.00	0.00	0.00

Tabela 4 – Comparação dos resultados para as redes Mux Tree, Ômega, Crossbar e Clos de tamanho 4096x32.

Rede	**Muxes**	δm	**Gates**	δg	**%Blq**	**%Min**	**%Max**	**%Amtr**
Mux Tree	4064	1.00	18201.69	1.00	36.06	12.50	62.50	100.00
Ômega G1	4224	1.03	18857.185	1.04	36.07	9.37	62.50	100.00
Ômega G2	4384	1.07	19512.597	1.07	21.29	0.00	53.12	99.99
Ômega G3	4576	1.12	20299.081	1.12	13.04	0.00	40.62	99.79
Ômega G4	4704	1.15	20823.404	1.14	11.76	0.00	40.62	99.42
Ômega G5	4992	1.22	22003.138	1.21	7.77	0.00	31.25	95.79
Ômega G6	5088	1.25	22396.389	1.23	6.82	0.00	31.25	93.56
Ômega G7	5248	1.29	23051.801	1.27	6.28	0.00	31.25	91.48
Ômega G8	5344	1.31	23445.034	1.29	6.32	0.00	31.25	91.14
Ômega G9	5856	1.44	25542.342	1.40	4.39	0.00	25.00	80.16
Ômega G10	5920	1.45	25804.504	1.42	3.81	0.00	21.87	75.10
Crossbar	-	-	80756.705	4.44	0.00	0.00	0.00	0.00
Clos R3	-	-	139170.151	7.65	0.00	0.00	0.00	0.00
Clos R5	-	-	178427.579	9.80	0.00	0.00	0.00	0.00
Clos R7	-	-	167458.388	9.20	0.00	0.00	0.00	0.00

Tabela 5 – Comparação dos resultados para as redes Mux Tree, Ômega, Crossbar e Clos de tamanho 512x64.

Rede	**Muxes**	δm	**Gates**	δg	**%Blq**	**%Min**	**%Max**	**%Amtr**
Mux Tree	448	1.00	2016.475	1.00	34.09	17.18	53.12	100.00
Ômega G1	832	1.85	3589.526	1.78	34.08	17.18	53.12	100.00
Ômega G2	1216	2.71	5162.511	2.56	17.70	3.12	35.93	100.00
Ômega G3	1664	3.71	6997.658	3.47	8.62	0.00	23.43	99.97
Ômega G4	1984	4.42	8308.465	4.12	6.94	0.00	21.87	99.78
Ômega G5	2624	5.85	10930.095	5.42	3.14	0.00	15.62	90.58
Ômega G6	2880	6.42	11978.758	5.94	1.63	0.00	10.93	70.80
Ômega G7	3264	7.28	13551.726	6.72	0.94	0.00	10.93	49.39
Ômega G8	3520	7.85	14600.388	7.24	0.75	0.00	9.37	40.95
Ômega G9	3442	7.68	14280.878	7.08	2.89	0.00	12.50	92.37
Ômega G10	3382	7.54	14035.093	6.96	5.18	0.00	15.62	99.55
Crossbar	-	-	20236.9	10.04	0.00	0.00	0.00	0.00
Clos R3	-	-	21398.531	10.61	0.00	0.00	0.00	0.00
Clos R5	-	-	18779.874	9.31	0.00	0.00	0.00	0.00
Clos R7	-	-	21674.826	10.75	0.00	0.00	0.00	0.00

Tabela 6 – Comparação dos resultados para as redes Mux Tree, Ômega, Crossbar e Clos de tamanho 1024x64.

Rede	**Muxes**	δm	**Gates**	δg	**%Blq**	**%Min**	**%Max**	**%Amtr**
Mux Tree	960	1.00	4308.913	1.00	35.33	17.18	56.25	100.00
Ômega G1	1344	1.40	5881.981	1.37	35.31	17.18	53.12	100.00
Ômega G2	1728	1.80	7454.95	1.73	19.66	4.68	39.06	100.00
Ômega G3	2176	2.26	9290.096	2.16	10.91	0.00	29.68	99.99
Ômega G4	2496	2.60	10600.903	2.46	9.50	0.00	26.56	99.97
Ômega G5	3136	3.26	13222.534	3.07	5.80	0.00	21.87	99.06
Ômega G6	3392	3.53	14271.196	3.31	4.34	0.00	18.75	96.89
Ômega G7	3776	3.93	15844.164	3.68	3.77	0.00	17.18	94.76
Ômega G8	4032	4.20	16892.826	3.92	3.75	0.00	15.62	94.36
Ômega G9	5120	5.33	21349.603	4.95	2.07	0.00	14.06	75.15
Ômega G10	5312	5.53	22136.088	5.14	1.48	0.00	12.50	63.00
Crossbar	-	-	41535.35	9.64	0.00	0.00	0.00	0.00
Clos R3	-	-	53303.997	12.37	0.00	0.00	0.00	0.00
Clos R5	-	-	46586.985	10.81	0.00	0.00	0.00	0.00
Clos R7	-	-	43498.43	10.09	0.00	0.00	0.00	0.00

Tabela 7 – Comparação dos resultados para as redes Mux Tree, Ômega, Crossbar e Clos de tamanho 2048x64.

Rede	**Muxes**	δm	**Gates**	δg	**%Blq**	**%Min**	**%Max**	**%Amtr**
Mux Tree	1984	1.00	8893.79	1.00	35.91	17.18	53.12	100.00
Ômega G1	2368	1.19	10466.858	1.18	35.92	18.75	53.12	100.00
Ômega G2	2752	1.38	12039.826	1.35	20.60	3.12	40.62	100.00
Ômega G3	3200	1.61	13874.973	1.56	12.02	0.00	32.81	99.99
Ômega G4	3520	1.77	15185.796	1.71	10.70	0.00	31.25	99.99
Ômega G5	4160	2.09	17807.427	2.00	6.99	0.00	23.43	99.65
Ômega G6	4416	2.22	18856.072	2.12	5.65	0.00	21.87	98.91
Ômega G7	4800	2.41	20429.057	2.30	5.12	0.00	20.31	98.17
Ômega G8	5056	2.54	21477.703	2.41	5.12	0.00	21.87	98.03
Ômega G9	6144	3.09	25934.48	2.92	3.46	0.00	17.18	91.71
Ômega G10	6336	3.19	26720.964	3.00	2.92	0.00	17.18	87.76
Crossbar	-	-	80988.042	9.11	0.00	0.00	0.00	0.00
Clos R3	-	-	104088.173	11.70	0.00	0.00	0.00	0.00
Clos R5	-	-	100788.892	11.33	0.00	0.00	0.00	0.00
Clos R7	-	-	88559.017	9.96	0.00	0.00	0.00	0.00

Tabela 8 – Comparação dos resultados para as redes Mux Tree, Ômega, Crossbar e Clos de tamanho 4096x64.

Rede	**Muxes**	δm	**Gates**	δg	**%Blq**	**%Min**	**%Max**	**%Amtr**
Mux Tree	4032	1.00	18063.543	1.00	36.21	17.18	54.68	100.00
Ômega G1	4416	1.09	19636.611	1.09	36.21	17.18	54.68	100.00
Ômega G2	4800	1.19	21209.596	1.17	21.06	4.68	40.62	100.00
Ômega G3	5248	1.30	23044.726	1.28	12.56	0.00	31.25	99.99
Ômega G4	5568	1.38	24355.549	1.35	11.28	0.00	29.68	99.99
Ômega G5	6208	1.53	26977.18	1.49	7.59	0.00	23.43	99.79
Ômega G6	6464	1.60	28025.825	1.55	6.28	0.00	21.87	99.35
Ômega G7	6848	1.69	29598.81	1.64	5.77	0.00	21.87	98.90
Ômega G8	7104	1.76	30647.456	1.70	5.77	0.00	21.87	98.83
Ômega G9	8192	2.03	35104.233	1.94	4.11	0.00	18.75	94.98
Ômega G10	8384	2.07	35890.717	1.99	3.58	0.00	17.18	92.57
Crossbar	-	-	161167.461	8.92	0.00	0.00	0.00	0.00
Clos R3	-	-	258535.948	14.31	0.00	0.00	0.00	0.00
Clos R5	-	-	225204.298	12.47	0.00	0.00	0.00	0.00
Clos R7	-	-	170798.425	9.46	0.00	0.00	0.00	0.00

Tabela 9 – Comparação dos resultados para as redes Mux Tree, Ômega, Crossbar e Clos de tamanho 1024x128.

Rede	**Muxes**	δm	**Gates**	δg	**%Blq**	**%Min**	**%Max**	**%Amtr**
Mux Tree	896	1.00	4032.618	1.00	34.22	21.87	47.65	100.00
Ômega G1	1792	2.00	7702.977	1.91	34.22	22.65	46.87	100.00
Ômega G2	2688	3.00	11373.27	2.82	17.57	6.25	30.46	100.00
Ômega G3	3712	4.14	15567.869	3.86	8.42	0.00	18.75	99.99
Ômega G4	4480	5.00	18713.839	4.64	6.63	0.00	17.96	99.99
Ômega G5	5888	6.57	24481.423	6.07	2.84	0.00	13.28	98.32
Ômega G6	6528	7.28	27103.054	6.72	1.42	0.00	8.59	87.62
Ômega G7	7424	8.28	30773.33	7.63	0.76	0.00	6.25	67.25
Ômega G8	8064	9.00	33394.977	8.28	0.59	0.00	6.25	57.02
Ômega G9	7908	8.82	32755.941	8.12	3.35	0.00	9.37	99.81
Ômega G10	7790	8.69	32272.591	8.00	5.79	0.00	13.28	99.99
Crossbar	-	-	82977.994	20.58	0.00	0.00	0.00	0.00
Clos R3	-	-	60881.417	15.10	0.00	0.00	0.00	0.00
Clos R5	-	-	48433.94	12.01	0.00	0.00	0.00	0.00
Clos R7	-	-	47544.402	11.79	0.00	0.00	0.00	0.00

Tabela 10 – Comparação dos resultados para as redes Mux Tree, Ômega, Crossbar e Clos de tamanho 2048x128.

Rede	**Muxes**	δm	**Gates**	δg	**%Blq**	**%Min**	**%Max**	**%Amtr**
Mux Tree	1920	1.00	8617.495	1.00	35.46	22.65	49.21	100.00
Ômega G1	2816	1.46	12287.87	1.43	35.46	22.65	47.65	100.00
Ômega G2	3712	1.93	15958.147	1.85	19.59	7.81	32.81	100.00
Ômega G3	4736	2.46	20152.762	2.34	10.78	0.78	21.87	100.00
Ômega G4	5504	2.86	23298.716	2.70	9.31	0.00	20.31	99.99
Ômega G5	6912	3.60	29066.3	3.37	5.63	0.00	15.62	99.98
Ômega G6	7552	3.93	31687.93	3.68	4.23	0.00	14.06	99.87
Ômega G7	8448	4.40	35358.223	4.10	3.62	0.00	12.50	99.64
Ômega G8	9088	4.73	37979.854	4.41	3.49	0.00	12.50	99.52
Ômega G9	11392	5.93	47417.714	5.50	2.01	0.00	9.37	92.94
Ômega G10	11904	6.20	49515.022	5.75	1.32	0.00	9.37	82.01
Crossbar	-	-	161802.245	18.78	0.00	0.00	0.00	0.00
Clos R3	-	-	114710.617	13.31	0.00	0.00	0.00	0.00
Clos R5	-	-	103063.26	11.96	0.00	0.00	0.00	0.00
Clos R7	-	-	95397.701	11.07	0.00	0.00	0.00	0.00

Tabela 11 – Comparação dos resultados para as redes Mux Tree, Ômega, Crossbar e Clos de tamanho 4096x128.

Rede	**Muxes**	δm	**Gates**	δg	**%Blq**	**%Min**	**%Max**	**%Amtr**
Mux Tree	3968	1.00	17787.248	1.00	36.05	23.43	49.21	100.00
Ômega G1	4864	1.22	21457.624	1.21	36.06	21.87	49.21	100.00
Ômega G2	5760	1.45	25127.9	1.41	20.55	8.59	37.50	100.00
Ômega G3	6784	1.70	29322.515	1.65	11.92	1.56	24.21	100.00
Ômega G4	7552	1.90	32468.469	1.83	10.54	0.00	23.43	99.99
Ômega G5	8960	2.25	38236.053	2.15	6.83	0.00	18.75	99.99
Ômega G6	9600	2.41	40857.683	2.30	5.54	0.00	16.40	99.98
Ômega G7	10496	2.64	44527.976	2.50	5.01	0.00	14.84	99.96
Ômega G8	11136	2.80	47149.607	2.65	4.92	0.00	14.06	99.94
Ômega G9	13440	3.38	56587.484	3.18	3.50	0.00	13.28	99.31
Ômega G10	13952	3.51	58684.791	3.30	2.83	0.00	11.71	98.17
Crossbar	-	-	321989.47	18.10	0.00	0.00	0.00	0.00
Clos R3	-	-	276132.185	15.52	0.00	0.00	0.00	0.00
Clos R5	-	-	228003.919	12.82	0.00	0.00	0.00	0.00
Clos R7	-	-	182844.413	10.28	0.00	0.00	0.00	0.00

Tabela 12 – Comparação dos resultados para as redes Mux Tree, Ômega, Crossbar e Clos de tamanho 1024x256.

Rede	**Muxes**	δm	**Gates**	δg	**%Blq**	**%Min**	**%Max**	**%Amtr**
Mux Tree	768	1.00	3480.028	1.00	31.57	23.04	39.84	100.00
Ômega G1	2816	3.66	11869.326	3.41	31.57	22.65	39.84	100.00
Ômega G2	4864	6.33	20258.557	5.82	12.79	5.07	21.09	100.00
Ômega G3	7168	9.33	29696.417	8.53	3.04	0.00	9.37	99.99
Ômega G4	8960	11.66	37036.986	10.64	0.90	0.00	5.07	95.03
Ômega G5	8754	11.39	36193.154	10.40	7.80	1.95	14.84	100.00
Ômega G6	8618	11.22	35636.063	10.24	10.71	4.29	17.57	100.00
Ômega G7	8594	11.19	35537.742	10.21	11.80	4.68	19.14	100.00
Ômega G8	8512	11.08	35201.84	10.12	13.89	7.42	21.09	100.00
Ômega G9	8452	11.00	34956.071	10.04	11.45	3.90	18.75	100.00
Ômega G10	8326	10.84	34439.936	9.90	12.96	6.25	20.70	100.00
Crossbar	-	-	165855.956	47.66	0.00	0.00	0.00	0.00
Clos R3	-	-	70978.342	20.40	0.00	0.00	0.00	0.00
Clos R5	-	-	56671.106	16.28	0.00	0.00	0.00	0.00
Clos R7	-	-	53528.159	15.38	0.00	0.00	0.00	0.00

Tabela 13 – Comparação dos resultados para as redes Mux Tree, Ômega, Crossbar e Clos de tamanho 2048x256.

Rede	**Muxes**	δm	**Gates**	δg	**%Blq**	**%Min**	**%Max**	**%Amtr**
Mux Tree	1792	1.00	8064.905	1.00	34.29	24.21	43.35	100.00
Ômega G1	3840	2.14	16454.202	2.04	34.29	25.39	43.35	100.00
Ômega G2	5888	3.28	24843.433	3.08	17.45	8.98	26.56	100.00
Ômega G3	8192	4.57	34281.294	4.25	8.17	2.34	16.01	100.00
Ômega G4	9984	5.57	41621.863	5.16	6.44	1.17	14.84	100.00
Ômega G5	13056	7.28	54205.693	6.72	2.66	0.00	8.98	99.89
Ômega G6	14592	8.14	60497.616	7.50	1.22	0.00	6.25	96.56
Ômega G7	16640	9.28	68886.831	8.54	0.60	0.00	4.68	81.59
Ômega G8	18176	10.14	75178.754	9.32	0.43	0.00	3.90	70.44
Ômega G9	17864	9.96	73900.699	9.16	3.74	0.00	8.20	99.99
Ômega G10	17618	9.83	72893.009	9.04	6.27	1.56	11.32	100.00
Crossbar	-	-	323420.189	40.10	0.00	0.00	0.00	0.00
Clos R3	-	-	127250.319	15.78	0.00	0.00	0.00	0.00
Clos R5	-	-	114703.974	14.22	0.00	0.00	0.00	0.00
Clos R7	-	-	104804.454	13.00	0.00	0.00	0.00	0.00

Tabela 14 – Comparação dos resultados para as redes Mux Tree, Ômega, Crossbar e Clos de tamanho 4096x256.

Rede	**Muxes**	δm	**Gates**	δg	**%Blq**	**%Min**	**%Max**	**%Amtr**
Mux Tree	3840	1.00	17234.658	1.00	35.53	25.39	44.53	100.00
Ômega G1	5888	1.53	25623.972	1.49	35.54	26.56	45.70	100.00
Ômega G2	7936	2.06	34013.186	1.97	19.54	10.54	28.90	100.00
Ômega G3	10240	2.66	43451.063	2.52	10.63	3.51	18.75	100.00
Ômega G4	12032	3.13	50791.632	2.95	9.17	2.73	17.18	100.00
Ômega G5	15104	3.93	63375.462	3.68	5.49	0.00	13.67	99.99
Ômega G6	16640	4.33	69667.369	4.04	4.07	0.00	10.54	99.99
Ômega G7	18688	4.86	78056.584	4.53	3.48	0.00	9.76	99.99
Ômega G8	20224	5.26	84348.507	4.89	3.32	0.00	9.37	99.99
Ômega G9	25088	6.53	104272.906	6.05	1.94	0.00	7.81	99.24
Ômega G10	26368	6.86	109516.168	6.35	1.22	0.00	5.85	94.63
Crossbar	-	-	643626.197	37.34	0.00	0.00	0.00	0.00
Clos R3	-	-	297387.52	17.26	0.00	0.00	0.00	0.00
Clos R5	-	-	249663.5	14.49	0.00	0.00	0.00	0.00
Clos R7	-	-	195437.71	11.34	0.00	0.00	0.00	0.00

APÊNDICE G – Detalhamento dos resultados do estudo de caso

Nesse apêndice são apresentados os detalhamentos dos resultados do estudo de caso da seção 4.2. Na Tabela 15, são apresentados os detalhamentos dos resultados para o projeto Amber (SANTIFORT, 2013). A Tabela 16 apresenta os detalhamentos para o projeto MIPS 32 Release 1 (AYERS, 2014). Esses dois projetos são processadores. O terceiro projeto é o módulo s38584 de *benchmark* do ISCAS (ALBRECHT; Cadence Research Laboratories, 2005), e os detalhamentos para esse projeto são apresentados na Tabela 17.

Os valores de área apresentados nos experimentos levam em consideração a área total do circuito com portas e fios. Esse valor é calculado pela ferramenta de síntese **Encounter RTL Compiler 11.25** (Cadence Design Systems, 2015). Para obter o valor em NANDs equivalentes, a área total é dividida pelo tamanho da maior célula NAND da biblioteca de células utilizada na síntese. Dessa forma, foi utilizado a célula NAND2 cujo tamanho é $8.96\mu m * 6.72\mu m = 60.211\mu m$ (IBM Systems and Technology Group, 2010, p. 48).

A primeira linha da tabela apresenta o resultado para o projeto. As linhas restantes apresentam o resultado do projeto com uma rede de interconexão instanciada dentro do projeto. Em cada tabela apresentada é realizado uma comparação entre as redes Mux Tree, Clos rearranjável de 3, 5 e 7 estágios (Clos R3, Clos R5 e Clos R7), e redes Ômega com diferentes arranjos nas portas de saída. Conforme foi apresentado na seção 3.2, a rede Ômega apresenta o sufixo Gi para diferenciar os arranjos das portas de saída. Ômega G1 representa uma rede cuja saída é agrupada no topo, como é mostrado na Figura 21d. A rede Ômega G2 terá disponíveis as saídas {0, 2, 4, 6, ...} (Figura 24b), e a rede Ômega G3 terá disponíveis as saídas {0, 3, 6, 9, 12, ...}.

Na coluna **Gates** são apresentados os custos de área em termos de NANDs equivalentes. **A primeira linha contém apenas o projeto** e é o referencial para avaliar o custo adicional. A coluna δa apresenta o custo de $(Gates_i)/(Gates_1)$, sendo i uma linha qualquer da tabela.

Tabela 15 – Resultados de síntese para o estudo de caso do projeto Amber. Comparação dos custos de área para as redes Mux Tree, Ômega e Clos de tamanho 100x25 e 256x32 instanciadas no projeto Amber.

Projeto	**Gates**	δg	**Projeto**	**Gates**	δg
Amber	7925.097	1.00	Amber	7925.097	1.00
Amber + 100x25 Mux Tree	8327.963	1.05	Amber + 256x32 Mux Tree	9035.591	1.14
Amber + 100x25 Omega G1	9160.834	1.16	Amber + 256x32 Omega G1	9143.163	1.15
Amber + 100x25 Omega G2	9701.682	1.22	Amber + 256x32 Omega G2	10217.568	1.29
Amber + 100x25 Omega G3	10121.555	1.28	Amber + 256x32 Omega G3	11004.052	1.39
Amber + 100x25 Crossbar	9833.917	1.24	Amber + 256x32 Crossbar	13654.797	1.72
Amber + 100x25 Clos R3	10683.994	1.35	Amber + 256x32 Clos R3	17257.859	2.18
Amber + 100x25 Clos R5	11313.746	1.43	Amber + 256x32 Clos R5	15749.663	1.99
Amber + 100x25 Clos R7	11491.355	1.45	Amber + 256x32 Clos R7	16360.997	2.06

Tabela 16 – Resultados de síntese para o estudo de caso do projeto MIPS 32. Comparação dos custos de área para as redes Mux Tree, Ômega e Clos de tamanho 100x25 e 256x32 instanciadas no projeto MIPS 32.

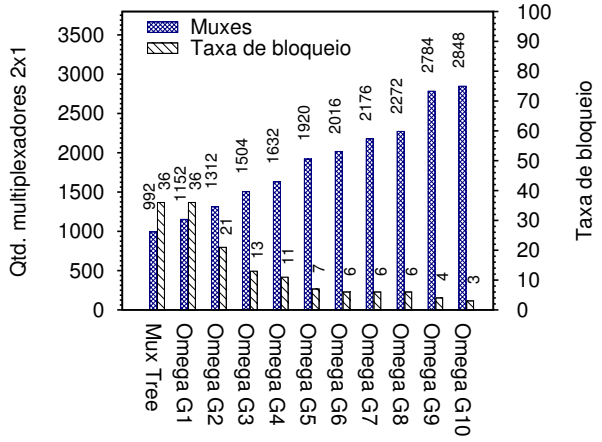
Projeto	**Gates**	δg	**Projeto**	**Gates**	δg
Mips32	12552.225	1.00	Mips32	12552.225	1.00
Mips32 + 100x25 Mux Tree	12891.116	1.03	Mips32 + 256x32 Mux Tree	13551.261	1.08
Mips32 + 100x25 Omega G1	13698.294	1.09	Mips32 + 256x32 Omega G1	14118.25	1.12
Mips32 + 100x25 Omega G2	14235.704	1.13	Mips32 + 256x32 Omega G2	14733.437	1.17
Mips32 + 100x25 Omega G3	14572.337	1.16	Mips32 + 256x32 Omega G3	15600.355	1.24
Mips32 + 100x25 Crossbar	13972.928	1.11	Mips32 + 256x32 Crossbar	18754.181	1.49
Mips32 + 100x25 Clos R3	15301.556	1.22	Mips32 + 256x32 Clos R3	21771.287	1.73
Mips32 + 100x25 Clos R5	15820.647	1.26	Mips32 + 256x32 Clos R5	20362.475	1.62
Mips32 + 100x25 Clos R7	15954.277	1.27	Mips32 + 256x32 Clos R7	20968.876	1.67

Tabela 17 – Resultados de síntese para o estudo de caso do projeto ISCAS s38584. Comparação dos custos de área para as redes Mux Tree, Ômega e Clos de tamanho 100x25 e 256x32 instanciadas no projeto ISCAS s38584.

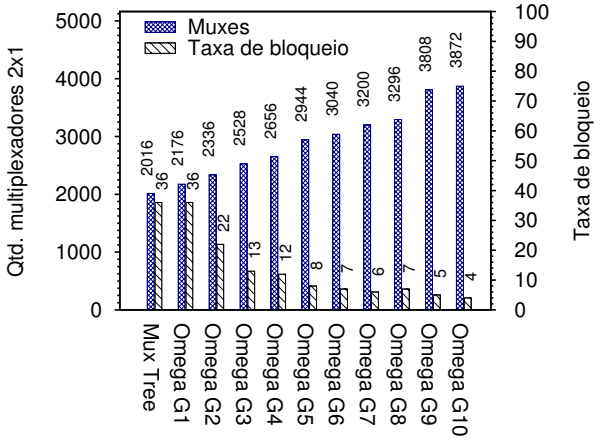
Projeto	**Gates**	δg	**Projeto**	**Gates**	δg
Iscas	4104.034	1.00	Iscas	4104.034	1.00
Iscas + 100x25 Mux Tree	4787.015	1.17	Iscas + 256x32 Mux Tree	5836.624	1.42
Iscas + 100x25 Omega G1	5591.187	1.36	Iscas + 256x32 Omega G1	6307.036	1.54
Iscas + 100x25 Omega G2	6132.949	1.49	Iscas + 256x32 Omega G2	6952.932	1.69
Iscas + 100x25 Omega G3	6507.515	1.59	Iscas + 256x32 Omega G3	7725.681	1.88
Iscas + 100x25 Crossbar	6372.074	1.55	Iscas + 256x32 Crossbar	13334.822	3.25
Iscas + 100x25 Clos R3	7179.236	1.75	Iscas + 256x32 Clos R3	13591.901	3.31
Iscas + 100x25 Clos R5	7837.554	1.91	Iscas + 256x32 Clos R5	12526.216	3.05
Iscas + 100x25 Clos R7	7977.761	1.94	Iscas + 256x32 Clos R7	13229.077	3.22

APÊNDICE H – Quantidade de multiplexadores versus taxa de bloqueio

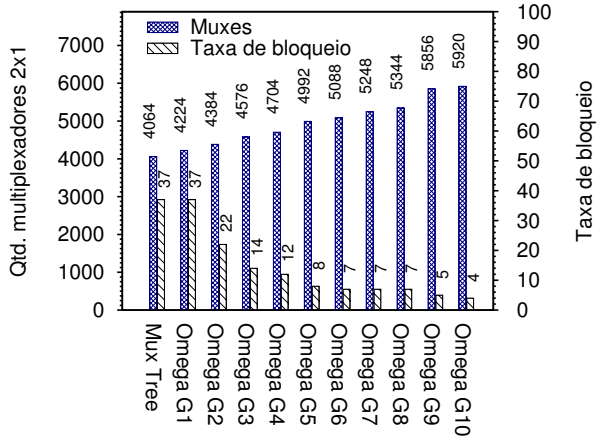
Nesse apêndice são apresentados comparativos da quantidade de multiplexadores 2x1 versus a taxa média de bloqueio para redes Ômega e Mux Tree. Conforme é apresentado na seção 4.1, essa taxa foi calculada através de experimentos: 1 milhão de amostras que formam conjuntos de portas de entrada para as redes. A medição do tamanho da rede somente pela quantidade de multiplexadores 2x1 descarta o custo do controle da rede. Conforme os dados de detalhamento dos resultados no Apêndice F, $\delta g \div \delta m$ apresenta variação de $\approx -1\%$ em uma rede 512x32 até $\approx -9\%$ em uma rede 4096x256.



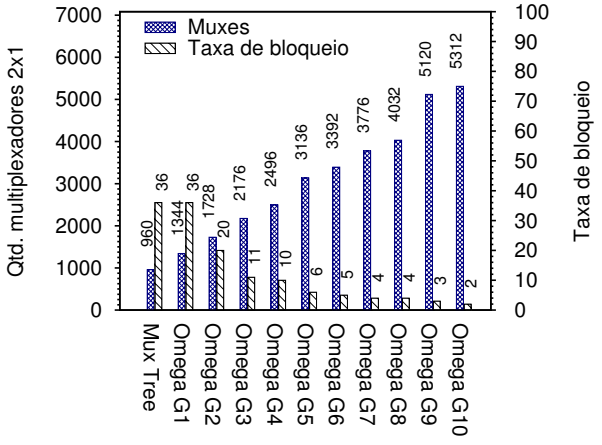
(a) 1024x32



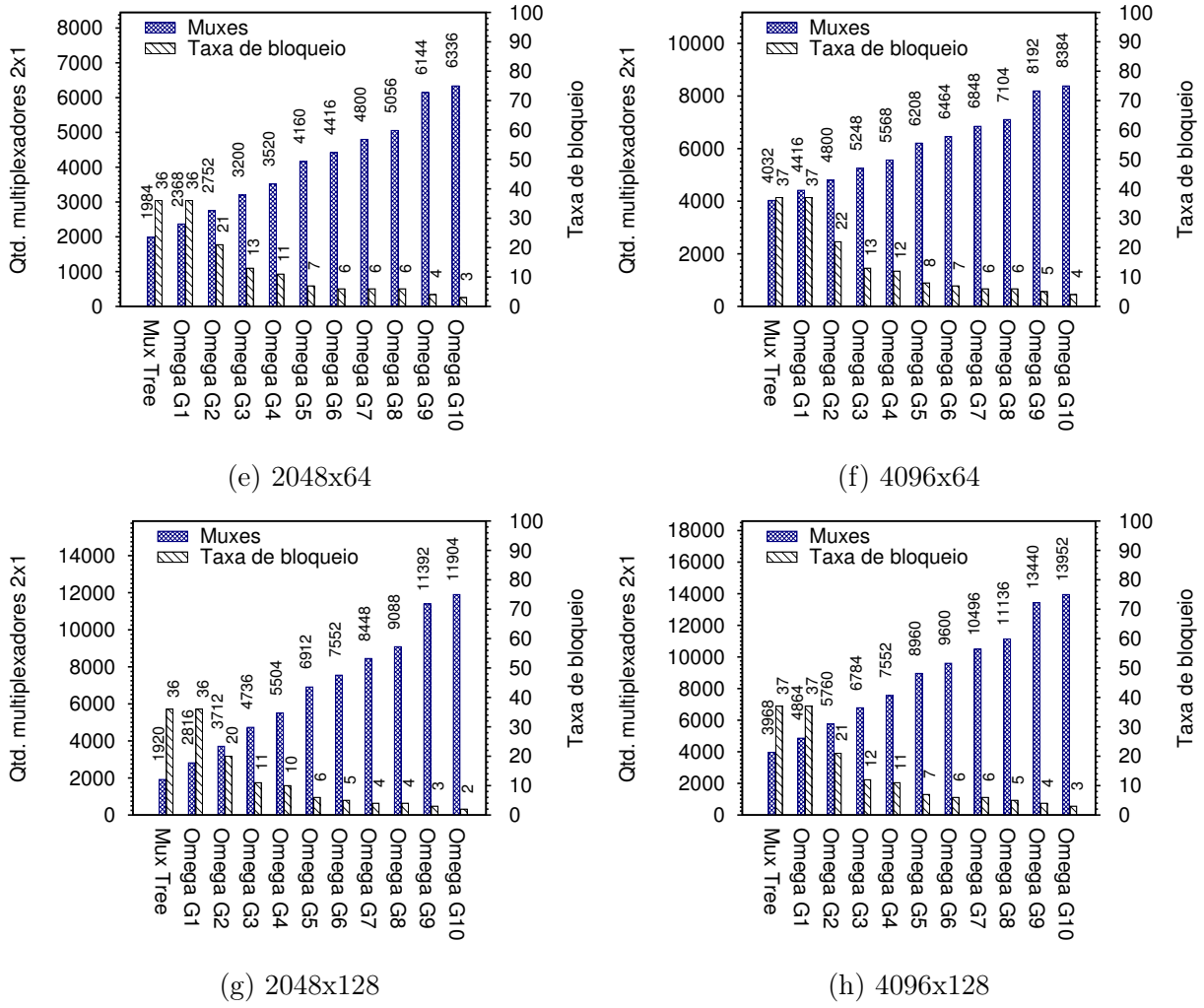
(b) 2048x32



(c) 4096x32



(d) 1024x64



(e) 2048x64

(f) 4096x64

(g) 2048x128

(h) 4096x128

Figura 57 – Comparação da quantidade de multiplexadores 2x1 e taxa média de bloqueio para várias redes Mux Tree e Omega.

APÊNDICE I – Utilização do Vericonn para gerar redes de interconexão

Vericonn é uma ferramenta de linha de comando capaz de gerar várias redes de interconexão simétricas e assimétricas. Para isso, basta invocar o programa utilizando os parâmetros exigidos. No caso de algum parâmetro requerido não ser fornecido, o programa imprime uma informação de erro e apresenta as opções ao usuário. Por exemplo, ao executar o programa sem nenhum argumento as opções de parâmetro são apresentadas, conforme mostra a Figura 58.

```
$ vericonn
the option '--network' is required but missing

Allowed options:

Generic options:
-h [ --help ]         Produce help message
-v [ --version ]      Display version information
-n [ --network ] arg  {crossbar|clos|mux_tree|omega|benes}
-i [ --in-size ] arg  Number of inputs.
-o [ --out-size ] arg Number of outputs.
-N [ --size ] arg     Number of inputs/outputs.
-l [ --load ] arg     Enable 2 phase crossbar (load, configure)
-a [ --address ] arg  Instantiate a decoder to translate address in cs port

Clos network options:
-t [ --type ] arg            Type of network {nonblocking|rearrangeable}
-s [ --stages ] arg          Number of network stages.

Omega network options:
-x [ --extra-stages ] arg Number of extra stages
--sort-output arg         Sort output: (1) grouped, (2-10) uniformly spaced
                          by the given factor
```

Figura 58 – Vericonn: opções de comando.

Algumas opções são genéricas e válidas para todas as redes. Por exemplo, toda vez que o programa é executado, é necessário especificar as opções rede (**--network**) e tamanho da rede (**--in-size**, **--out-size**, **--size**). O tamanho pode ser especificado separadamente para a entrada e a saída, ou com um único parâmetro para redes simétricas. A opção **--load** pode ser especificado para gerar redes com a opção de *load/config*,

conforme é apresentado na seção 3.3.1. A opção **--address** pode ser especificada para habilitar a tradução de endereços conforme é apresentado na seção 3.3.2.

A rede Crossbar é a rede mais simples gerada pela ferramenta. Essa rede também é o elemento base para todas as demais redes. Para gerar essa rede basta invocar o programa conforme é mostrado na Figura 59. O programa irá gerar um arquivo Verilog cada módulo da rede. No exemplo apresentado, o programa gera uma rede de 10 entradas e 2 saídas. O programa gera cinco arquivos: **DSM10x2.v**, **Decoder2.v**, **IOCell10.v**, **Switch10.v** e **SwitchMatrix10x2.v**. Um exemplo de um módulo Crossbar é apresentado no Apêndice A.

```
$ vericonn --network crossbar --in-size 10 --out-size 2
DSM10x2.v
Decoder2.v
IOCell10.v
Switch10.v
SwitchMatrix10x2.v
```

Figura 59 – Vericonn: Geração de uma rede Crossbar.

Para gerar redes Clos deve ser especificado se a rede é rearranjável ou não bloqueante. Essa especificação é feito pelo parâmetro **--type**. O número de estágios é especificado pelo parâmetro **--stages**, e deve ser um número ímpar maior ou igual a 3. Apesar do Vericonn não restringir o número máximo de estágios, ferramentas podem ser incapazes de realizar a síntese. A Figura 60 apresenta a execução do Vericonn para gerar uma rede 36x12 rearranjável com 5 estágios. Nesse exemplo, são geradas 3 diferentes redes

```
$ vericonn -n clos -t rearrangeable -i 36 -o 12 -s 5
Clos36x12.v
Clos12x6.v
DSM3x2.v
DSM3x3.v
DSM4x3.v
Decoder2.v
Decoder3.v
IOCell3.v
IOCell4.v
Switch3.v
Switch4.v
SwitchMatrix3x2.v
SwitchMatrix3x3.v
SwitchMatrix4x3.v
```

Figura 60 – Vericonn: Geração de uma rede Clos 36x12 rearranjável de 5 estágios.

Crossbars (DSM). O estágio de entrada é composto por 12 módulos **DSM3x3**, o estágio intermediário é composto por 3 módulos **Clos12x6**, e o estágio de saída é composto por 6 módulos **DSM3x2**. A rede Clos utilizada no estágio intermediário, é composta por 4 **DSM3x3** no primeiro estágio, 3 **DSM4x3** no segundo estágio, e 3 **DSM3x2** no terceiro estágio.

Benes é um caso especial da rede Clos e possui o número de estágio já predefinido: $2log_2N - 1$. Assim, o parâmetro **--stages** não é aceito para redes Benes. Um exemplo de rede Benes é apresentado na Figura 61.

```
$ vericonn --network benes -N 32
Benes32x32.v
DSM2x2.v
Decoder2.v
IOCell2.v
Switch2.v
SwitchMatrix2x2.v
```

Figura 61 – Vericonn: Geração de uma rede Benes 32x32.

A rede Mux Tree é a única rede gerada pelo Vericonn que é uma rede exclusivamente assimétrica. Portanto, para a rede Mux Tree, a opção **--in-size** deve obrigatoriamente ser maior do que **--out-size**. As redes Mux Tree também são construídas pelo Vericonn utilizando módulos Crossbars. Conforme apresenta a Figura 62, para gerar uma rede Mux Tree de tamanho 11x2, a ferramenta Vericonn utiliza dois módulos DSM3x1. Essa situação é ilustrado na Figura 30 na seção 3.3.2.2.

```
$ vericonn -n mux_tree -i 11 -o 2
Mux_Tree11x2.v
DSM2x1.v
DSM3x1.v
Decoder1.v
IOCell2.v
IOCell3.v
Switch2.v
Switch3.v
SwitchMatrix2x1.v
SwitchMatrix3x1.v
```

Figura 62 – Vericonn: Geração de uma rede de Árvore de Multiplexadores 11x2.

Ao gerar uma rede Ômega, algumas redes Crossbars 2x2 são transformadas em Crossbars 2x1. Isso ocorre pelo durante o processo de remoção de multiplexadores apresentado no Capítulo 3. Essa situação é exemplificada na Figura 63. Nesse exemplo, a

ferramenta é executada com os parâmetros da rede e tamanho de entrada e saída. Outros dois parâmetros opcionais estão disponíveis para a geração de redes Ômega: **--sort-output** e **--extra-stages**. A opção **--sort-output** é utilizada para dispersar as saídas da rede Ômega, conforme é descrito no Capítulo 3. A opção **--extra-stages** é utilizada para incluir estágios extras na rede Ômega.

```
$ vericonn --network omega --in-size 32 --out-size 4
Omega32x4.v
DSM2x1.v
DSM2x2.v
Decoder1.v
Decoder2.v
IOCell2.v
Switch2.v
SwitchMatrix2x1.v
SwitchMatrix2x2.v
```

Figura 63 – Vericonn: Geração de uma rede de Ômega assimétrica 32x4.