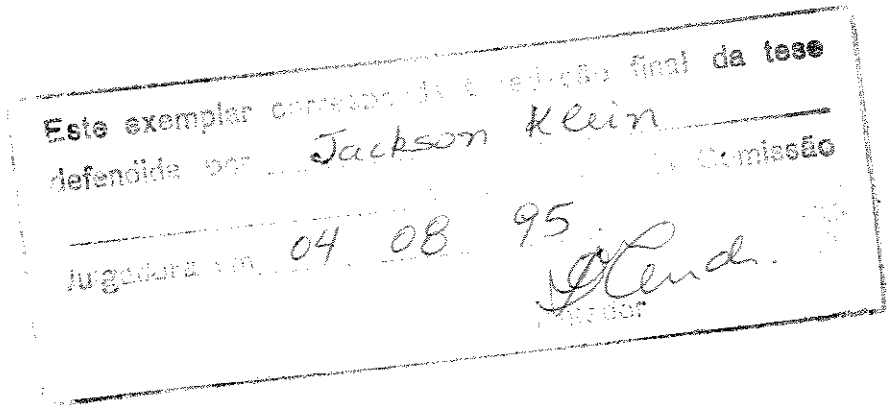
UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA ELÉTRICA
DEPARTAMENTO DE COMUNICAÇÕES



SIMNT: Uma Ferramenta para a Simulação de Sistemas de Comunicação

Tese submetida à Faculdade de Engenharia Elétrica da Universidade Estadual de Campinas, Departamento de Comunicações, como parte dos requisitos exigidos para a obtenção do título de

Mestre em Engenharia Elétrica.

Autor

Jackson Klein
Engenheiro Eletricista pela UFSM em 1993

Orientador

Prof. Dr. Leonardo de Souza Mendes
PhD em Engenharia Elétrica Syracuse University em 1992

Campinas, 04 de agosto de 1995.

Caixa Postal 6101, CEP 13083-970 Campinas - SP. Telefones: (0192) 39-8324/ 39-7502; Fax: (0192) 39-1395

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

K672s

Klein, Jackson
SIMNT : uma ferramenta para a simulação de sistemas de comunicação / Jackson Klein.--Campinas, SP: [s.n.], 1995.

Orientador: Leonardo de Souza Mendes.
Dissertação (mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica.

1. Sistemas de comunicação. 2. Simulação (Computadores digitais). 3. Comunicações óticas. I. Mendes, Leonardo de Souza. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica. III. Título.

Conteúdo

AGRADECIMENTOS

Agradeço a todas as pessoas que colaboraram durante a realização deste trabalho. Em especial:

Ao amigo Leonardo Mendes, pela confiança;

Aos colegas do DECOM e DMO;

e, principalmente...TINA.

Este trabalho contou com o apoio financeiro do CNPq.

ABSTRACT

In this work we present the SIMNT: a CAD-CAM (Computer Aided Design-Computer Aided Modeling) tool for the analysis and the design of communication systems. This tool uses modern simulation techniques such as modular structures, open topology, integration of devices and systems, definition of feedback blocks, a programmable library of models and automatic ordination and execution. To illustrate the use of the SIMNT, we present some examples of simulation, including a WDM optical link.

SUMÁRIO

Neste trabalho apresentamos o SIMNT: uma ferramenta de CAD-CAM (Computer Aided Design-Computer Aided Modeling) para a análise e o projeto de sistemas de comunicações. Esta ferramenta utiliza técnicas modernas de simulação tais como: estrutura modular, topologia aberta, integração de dispositivos e sistemas, definição de blocos realimentados, biblioteca de modelos programável e ordenação e execução automáticas. Para ilustrar o uso do SIMNT, apresentamos alguns exemplos de simulação, incluindo um enlace óptico WDM.

Lista de Figuras

Introdução

Os sistemas de comunicação atuais apresentam um constante aumento de complexidade e usualmente incluem alguns dispositivos não-lineares e fontes de ruído não-gaussianas[1][2]. Isto gera um aumento na dificuldade de avaliarmos seu desempenho analiticamente.

A análise e o modelamento de sistemas e dispositivos de comunicação podem ser realizados de diversas formas. A primeira forma é a experimental, utilizando-se laboratórios de pesquisa e desenvolvimento específicos. Esta abordagem, entretanto, consome muito tempo e é altamente dispendiosa. Além disso, os laboratórios são mais ou menos inflexíveis ao desenvolvimento de novas tecnologias. Uma segunda forma é através de técnicas de modelamento analítico, entretanto, formas fechadas para todos os modelos dos dispositivos de um sistema de comunicação somente são obtidas através da introdução de diversas simplificações e restrições. Isto limita sobremaneira o uso desta abordagem, muitas vezes tornando não-realísticos os sistemas sendo analisados. Uma terceira forma está no uso de CAD (Computer-Aided-Design) para o modelamento e análise dos dispositivos ou sistemas. Neste caso a simulação é flexível, permitindo a inclusão das características específicas de cada dispositivo, e uma análise mais realística dos sistemas. Todos estes fatores, aliados ao rápido desenvolvimento de computadores digitais, aumentaram o interesse na simulação de sistemas de comunicação[3][4][5][6][7].

Neste trabalho apresentamos o SIMNT, um aplicativo para desenvolvimento e análise de dispositivos e sistemas de comunicação em geral. Esta ferramenta faz uso de técnicas recentes de simulação, tais como estrutura modular, topologia aberta,

integração de dispositivos e sistemas, definição de blocos realimentados, biblioteca de modelos programável, ordenação e execução. O SIMNT suporta uma definição hierárquica dos blocos, ou seja, novos blocos podem ser definidos a partir dos já existentes, além disso, múltiplas ocorrências do mesmo bloco são admitidas.

A implementação do SIMNT surgiu da necessidade de se ter à disposição um aplicativo que permita a simulação de sistemas de comunicações (em particular sistemas de comunicações ópticas) e forneça recursos para o desenvolvimento de novos modelos de dispositivos. O SIMNT é desenvolvido em C++, e com isto usa a programação orientada para objetos. A utilização dos recursos de simulação do programa dispensa o conhecimento da linguagem C++, porém, os desenvolvedores de novos modelos devem conhecer esta linguagem, para a manipulação das funções de entrada e saída de dados. O SIMNT é desenvolvido para o sistema operacional WINDOWS NT, que apresenta recursos de programação avançada, tais como processamento distribuido e multienlace[8].

O princípio de funcionamento do SIMNT é similar ao programa BLOSIM[9], que foi inicialmente desenvolvido em C, mas, para utilizar os recursos de programação orientada para objetos, foi novamente implementado em LISP[6]. No SIMNT uma interface gráfica amigável servirá de ambiente de trabalho para o usuário gerar os arquivos de topologia e parâmetros para o sistema a ser simulado. Na primeira versão o usuário deve fornecer estes arquivos ao simulador. O SIMNT interpreta um sistema de comunicação como um conjunto de blocos interligados, os quais representam dispositivos e métodos numéricos.

Esta dissertação é dividida em quatro capítulos. O primeiro capítulo apresenta os recursos de programação empregados no desenvolvimento do SIMNT, descrevendo os

fundamentos da programação orientada para objetos e o sistema operacional Windows NT. O segundo capítulo apresenta a estrutura modular do programa, descrevendo a implementação e hierarquia dos objetos que o compõem. O capítulo três descreve o funcionamento do SIMNT, bem como os passos necessários para sua utilização na simulação e projeto de um sistema específico. Finalmente, o capítulo quatro demonstra a aplicação do programa para simulação de sistemas de comunicação óptica. Neste capítulo foram escolhidos sistemas de relativa complexidade estrutural, de modo a utilizar todos os recursos do SIMNT.

Referências Bibliográficas

[1] P. Lagasse, "Use and Limitations of Modeling Tools for Lightwave Material Devices and Systems", *ECOC'90* ,pp 789-830, 1990.

[2] A. Elrefaie, "Computer-Aided Modeling, and Simulation of Lightwave Communication Systems", *LEOS Newsletter*, vol 7, No. 1, pp 1,19-20, fevereiro de 1993.

[3] W. H. Tranter, K. L. Kosbar, "Simulation of Communication Systems", *IEEE Communications Magazine*, pp 26-35, julho de 1994.

[4] D. G. Duff, "Computer-Aided Design of Digital Lightwave Systems", *IEEE Journal on Selected Areas in Communications*, Vol. SAC-2, No. 1, pp171-185, janeiro de 1984.

[5] A. Elrefaie, J. K. Townsend, M. B. Romeiser, K. S Shanmugan, "Computer Simulation of Digital Lightwave Links", *IEEE Journal on Selected Areas in Communications*, Vol. 6, No. 1, pp 94-105, janeiro de 1988.

[6] K. S. Shanmugan, "An Update on Software Packages for Simulation on Communication Systems (Links) ", *IEEE Journal on Selected Areas in Communications*, Vol. 6, No. 1, pp 5-12, janeiro de 1988.

[7] K. S. Shanmugan, "Simulation and Implementation Tool for Signal Processing and Commnunication Systems", *IEEE Communications Magazine*, pp 36-40, julho de 1994.

[8] B. Myers, E. Hamer, *Dominando a Programação no Windos NT*, LTC-Livros Técnicos e Científicos , 1994.

[9] D. G. Messerschmitt, "A Tool for Structured Functional Simulation", *J. Select. Areas in commun.*, vol SAC-2, pp. 8-29, janeiro, 1984.

Capítulo 1

Recursos de Programação

1.1 Introdução

O SIMNT foi desenvolvido em linguagem de programação C++[1][2] para o sistema operacional de 32 bits Windows NT[3][4]. O C++ e as técnicas de programação orientada para objetos são conhecidas por facilidade de uso e simplicidade de manutenção. Este capítulo apresenta os recursos de programação utilizados na implementação do SIMNT, além de uma descrição sucinta do Windows NT e seus avançados recursos, dentre estes, multitarefa preemptiva e processamento distribuído.

1.2 Programação Orientada para Objetos: Um Sumário

Segundo Booch[5], a programação orientada para objetos (OOP) é definida como um método de implementação onde programas são organizados em coleções cooperativas de objetos, onde cada objeto representa uma ocorrência de uma classe, e onde algumas destas classes são membros de uma hierarquia de classes unidas por uma relação de herança. OOP é um método relativamente novo para projeto e implementação de software, e centraliza-se em torno de alguns conceitos básicos como: tipos de dados e classes abstratas, hierarquia de tipos, herança e polimorfismo.

Um **tipo abstrato de dado** engloba um tipo e um conjunto de operações associadas, operações estas que definem e caracterizam o seu comportamento.

Uma **definição de classe** descreve o comportamento de um tipo de dado abstrato definindo a interface para todas operações que podem ser realizadas e detalhes de implementação da estrutura de dados do mesmo. Quando detalhes de implementação são acessíveis somente no escopo da classe, este é um tipo *privado*, quando partes do dado são acessíveis fora do escopo da classe o tipo é *público*.

As operações relacionadas com o tipo também são divididas em públicas e privadas, estas operações são denominadas *métodos*. Estes métodos são análogos aos procedimentos e funções de linguagens não orientadas para objetos.

Um *objeto* é uma variável declarada como uma classe específica. Ele possui uma cópia de todos os dados e métodos definidos nesta classe. Isto permite que ações possam ser realizadas com o objeto simplesmente invocando estes métodos. Este processo é realizado enviando-se *mensagens* para o objeto. Uma mensagem contém parâmetros que são fornecidos aos dados que compõem o objeto. Cada variável, ou objeto, associada a uma classe representa uma ocorrência desta classe. Diversos objetos podem ser definidos como da mesma classe, ainda que possuam conjuntos de dados diferentes uns dos outros.

A OOP fornece uma *hierarquia* de tipos através de subclasses. Uma subclasse define seu comportamento específico adicionando novos métodos e dados àqueles herdados de objetos que hierarquicamente formam a base da classe. Devido ao *polimorfismo* estes métodos podem responder de diferentes maneiras a uma mesma mensagem numa hierarquia de classes.

Esta abordagem torna-se menos confusa observando-se a figura 1.1, onde apresentamos a hierarquia de classes do SIMNT; esta figura mostra a classe biblioteca, que forma a classe base, e suas classes derivadas, dadas respectivamente, pelas classes blocos, parâmetros, buffer, usuário e gerenciador. Daremos aqui uma breve descrição

das classes que compõem o programa. No próximo capítulo descreveremos detalhadamente a implementação deste programa.

O principal elemento que compõe a biblioteca é um vetor de objetos, cujo tamanho é definido pelo número de modelos disponíveis, tais como laser, fibra, amplificador etc. Cada objeto da classe biblioteca representa um modelo, implementado com um método, e suas características (nome, número de entradas, número de saídas). Quando um usuário desenvolve um modelo ele adiciona um novo objeto à classe biblioteca. A classe derivada *blocos* possui um vetor de objetos que formam os blocos do sistema. Cada bloco está associado a um dos objetos da classe biblioteca e adiciona novas características a esta classe (nome do bloco, arquivo de parâmetros, suas ligações com outros blocos, endereço para escrita nos buffers de saída, e diversas variáveis de estado). Os elementos principais da classe *parâmetros* são as *matrizes de parâmetros*. A cada bloco esta associado uma matriz de parâmetros. Um exemplo de um parâmetro do dispositivo amplificador é o ganho.

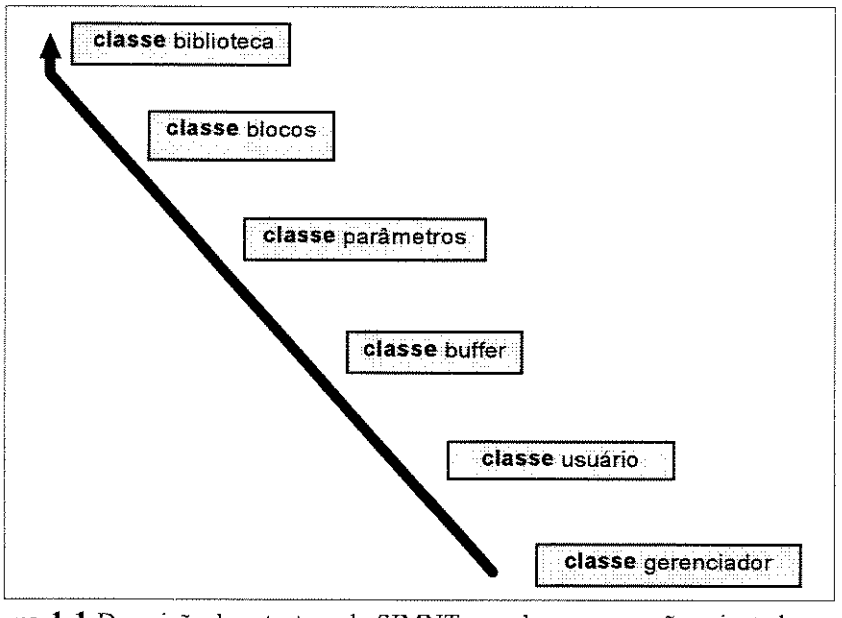


Figura 1.1 Descrição da estrutura do SIMNT usando programação orientada para objetos.

A classe *buffer* manipula vetores com as amostras, e suas características, que estão sendo processadas durante a simulação. Estas amostras possuem características tais como: frequência de amostragem, frequência central, tempo inicial, tamanho, domínio (tempo ou frequência), tipo (complexo, inteiro ou real), etc. As duas últimas classes são a classe *usuário*, formada por todos os métodos utilizados para implementação de modelos, e finalmente, a classe *gerenciador*, que é formada por métodos que ordenam e executam todo o processo de simulação. A classe gerenciador incorpora todas as propriedades das classes anteriores por herança e adiciona novos métodos para tornar seu comportamento específico.

Mas, como estes objetos se relacionam com um determinado bloco durante o processo de simulação? Na figura 1.2 temos um sistema óptico de seis blocos. Este sistema serve como exemplo para demonstrar os principais elementos que são associados a cada uma das classes. Vamos supor que apenas três modelos estejam disponíveis para simulação. Da figura 1.3 até a figura 1.8 indicamos o estado interno dos dados pertencentes a cada classe. São apresentados apenas os principais elementos que as compõem, pois a classe biblioteca, por exemplo, possui mais de 80 métodos, além de outros objetos.

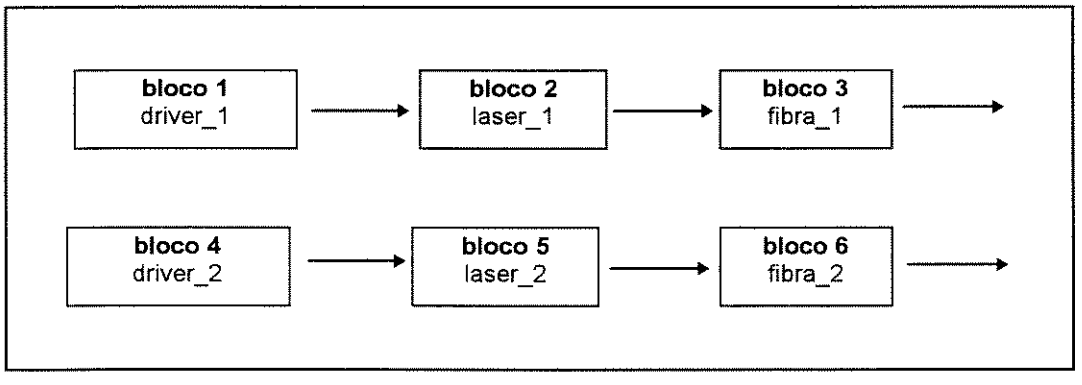


Figura 1.2 Sistema de blocos.

A classe biblioteca, representada na figura 1.3, independe do número de blocos do sistema, pois todos serão ocorrências dos modelos *driver_corrente*, *laser_monomodo* e *fibra_monomodo*, implementados com os métodos públicos *DriverNRZ*, *SMLaser1* e *OptFiber1*, respectivamente.

classe biblioteca

⇒ **objeto** 1 -	número de entradas	0
	número de saídas	1
	método	DriverNRZ
	nome	driver_corrente
⇒ **objeto** 2 -	número de entradas	1
	número de saídas	1
	método	SMlaser1
	nome	laser_monomodo
⇒ **objeto** 3 -	número de entradas	1
	número de saídas	1
	método	OptFiber1
	nome	fibra_monomodo

⇒ demais métodos para manipulação destes objetos.

Figura 1.3 Elementos da classe biblioteca.

A classe blocos herda os métodos e objetos da classe biblioteca, podendo manipular estes elementos, e adicionar novos. Conforme a figura 1.4, observamos que os blocos possuem nomes diferentes, mas sempre associados a um dos três dispositivos da classe biblioteca, ou seja, temos duas ocorrências de cada dispositivo. Cada objeto possui os seguintes dados: nome, dispositivo, ligações. Além disso herda outros dados da classe base, como por exemplo, número de entradas e saídas, necessários para verificação de erros de topologia.

A classe parâmetros, figura 1.5, associa a cada ocorrência de um bloco seus parâmetros específicos, por exemplo, o laser tem como parâmetros quantidades como: velocidade de grupo, fator de supressão do ganho, etc.

classe blocos

⇒ **objeto** 1 -	nome do dispositivo	driver_corrente
	nome do bloco	driver_1
	saída	bloco 2
⇒ **objeto** 2 -	nome do dispositivo	laser_monomodo
	nome do bloco	laser _1
	saída	bloco 3
	entrada	bloco 1
⇒ **objeto** 3 -	nome do dispositivo	fibra_monomodo
	nome do bloco	fibra_1
	entrada	bloco 2
⇒ **objeto** 4 -	nome do dispositivo	driver_corrente
	nome do bloco	driver_2
	saída	bloco 5
⇒ **objeto** 5 -	nome do dispositivo	laser_monomodo
	nome do bloco	laser _2
	saída	bloco 6
	entrada	bloco 4
⇒ **objeto** 6 -	nome do dispositivo	fibra_monomodo
	nome do bloco	fibra_2
	entrada	bloco 5

⇒ demais métodos para manipulação destes objetos.

Figura 1.4 Elementos da classe blocos.

classe parâmetros

⇒ **objeto** 1 - nome do bloco driver_1
parâmetros específicos: corrente de polarização, corrente de modulação, etc.

⇒ **objeto** 2 - nome do bloco laser _1
parâmetros específicos: velocidade de grupo, fator de supressão do ganho, etc.

⇒ **objeto** 3 - nome do bloco fibra_1
parâmetros específicos: atenuação, dispersão, etc.

⇒ **objeto** 4 - nome do bloco driver_2
parâmetros específicos: corrente de polarização, corrente de modulação, etc.

⇒ **objeto** 5 - nome do bloco laser _2
parâmetros específicos: velocidade de grupo, fator de supressão do ganho, etc.

⇒ **objeto** 6 - nome do bloco fibra_2
parâmetros específicos: atenuação, dispersão, etc.

⇒ demais métodos para manipulação destes objetos.

Figura 1.5 Elementos da classe parâmetros.

A classe buffer, figura 1.6, cria um objeto para cada saída do sistema. O número total de saídas é determinado pelas propriedade herdadas das classes anteriores. No nosso exemplo temos uma saída para cada bloco.

A classe usuário, figura 1.7, possui os métodos utilizados na classe base biblioteca; deste modo em uma única classe estão reunidas todas as funções para implementação dos modelos, mais especificamente, as funções de interface para entrada e saída de dados para os blocos.

classe buffer

⇒ **objeto** 1 - buffer saída do bloco 1
características: frequência de amostragem, frequência central, etc.

⇒ **objeto** 2 - buffer saída do bloco 2
características: frequência de amostragem, frequência central, etc.

⇒ **objeto** 3- buffer saída do bloco 3
características: frequência de amostragem, frequência central, etc.

⇒ **objeto** 4 - buffer saída do bloco 4
características: frequência de amostragem, frequência central, etc.

⇒ **objeto** 5 - buffer saída do bloco 5
características: frequência de amostragem, frequência central, etc.

⇒ **objeto** 6 - buffer saída do bloco 6
características: frequência de amostragem, frequência central, etc.

⇒ demais métodos para manipulação destes objetos.

Figura 1.6 Elementos da classe buffer.

classe usuário

⇒ **métodos** para implementação dos modelos da biblioteca.
⇒ demais métodos para manipulação destes objetos.

Figura 1.7 Elementos da classe usuário.

Finalmente, a classe gerenciador herda todos os métodos e objetos das demais classes. Deste modo pode esta classe administrar o processo de simulação, ou seja, pode determinar com a classe blocos, o nome do bloco1 (*driver_1*) e o dispositivo associado

(*driver_corrente*); pode obter com a classe biblioteca o seu número de saídas (1); pode, com a classe buffer, alocar espaço para esta saída (*objeto 1*); e pode, finalmente, com a classe parâmetros associar as características do dispositivo ao bloco. Uma de suas funções é definir a ordem de execução dos blocos. Na figura 1.8, a primeira ordenação é apresentada.

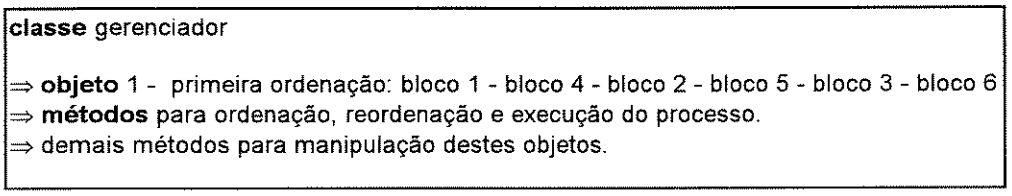


Figura 1.8 Elementos da classe gerenciador.

Diversas linguagens possuem recursos de programação orientada para objetos[5], tais como LISP, Smalltalk, Object Pascal, Ada ou C++. Pela sua grande popularidade e vasta bibliografia disponível, optamos pelo C++. Além disso, dispomos de produtos como o Borland C++[6][7] ou o Microsoft Visual C++[8], que possuem ambientes de desenvolvimento integrado com recursos de depuração para o Windows NT.

Decidimos por implementar a versão inicial do SIMNT, utilizando o Borland C++ 4.0, pois obedece ao *C++ Manual de Referência Comentado*[2]- documento oficial do ANSI, que irá servir como ponto de partida para padronização da linguagem. O C++ é uma linguagem de programação de uso geral baseada na linguagem C[9]. Além das vantagens fornecidas por C, o C++ também permite o uso de classes, funções inline, sobrecarga de operador, sobrecarga do nome de função, tipos de constantes, referências, operadores de gerenciamento livre, e verificação de argumentos de função e conversão de tipo. O uso de classes torna possível o uso das potencialidades da programação

orientada para objetos. Na lista 1.1 são apresentados alguns membros que compõe a classe *BlockLib*, a mesma classe biblioteca citada no início desta seção. A classe BlockLib contém uma sequência de objetos de variados tipos (linhas 6,7,8,9 e 11), um conjunto de funções (métodos) para manipulação destes objetos (linhas 13,17, 21 etc.), e um conjunto de restrições em relação ao acesso a esses objetos e funções (linhas 3,15 e 28).

```
1   class BlockLib
2       {
3           private:
4               struct StructLib
5                   {
6                       char  FunctionName[64];
7                       int  Nin;
8                       int  Nout;
9                       void  (BlockLib::*FunctionLib)();
10                  };
11              static StructLib   *SystemLib;
12
13              int GetBlockLibIndex(char*);
14
15          protected:
16
17              void AddLib(char *,void (BlockLib::*)(),int ,int);
18
19              void SetLibIndex(int);
20
21              int GetLibIndex(void);
22              char *GetFunctionName(void);
23              int GetLibNin(void);
24              int GetLibNout(void);
25
26              void Run(void);
27
28          public:
29
30              virtual void * ReadSample(int, long)=0;
31
32              void DriverNRZ(void);
33              void SMLaser1(void);
34              void OptFiber1(void);
35      };
```

Lista 1.1 Alguns membros da classe BlockLib.

1.3 O Sistema Operacional Windows NT

O Windows NT é um sistema operacional da Microsoft que executa processamento em 32 bits, o que aumenta substancialmente a velocidade das computações e do acesso a memória. Suporta múltiplas interfaces de programação de aplicativo, como o UNIX baseado em POSIX, MS-DOS, OS/2, e Windows (tabela 1.1 [4]). O Windows NT pode rodar muitos programas ao mesmo tempo, alternando rapidamente entre eles, executando a multitarefa preemptiva. Além disso, um único programa pode executar várias tarefas simultaneamente dividindo-se em enlaces. Os enlaces são executados independentemente, assim como os programas, mas vários deles podem compartilhar recursos e dados globais de um único programa. O Windows NT incorpora um gerenciamento de memória sofisticado, usando endereços virtuais para evitar que os programas interfiram uns com os outros. Este sistema operacional também possibilita a realização de processamento distribuído, permitindo que um programa divida seu trabalho entre máquinas diferentes. O processamento distribuído torna o poder da CPU efetivamente um recurso de rede a ser compartilhado tão facilmente quanto arquivos e impressoras.

Como o Windows NT é um sistema de multitarefa preemptiva, pode interromper os enlaces arbitrariamente e obrigar um enlace ou programa mal escrito a submeter-se ao seguinte, restabelecendo o fluxo do processamento. A multitarefa não-preemptiva do Windows 3.1 de 16 bits só permite a troca quando um processo dá uma pausa para uma nova mensagem do sistema. No Windows NT, um programa não pode congelar o sistema negligenciando ou recusando-se a receber sua próxima mensagem. Além disso, como a colocação de dados na memória é controlada pelo sistema de memória virtual, o espaço de endereçamento de cada processo pode ser separado e protegido dos outros.

	DOS	Windows 16	POSIX	OS/2	Windows NT
Multitarefa		(não-preemptiva)	x	x	x
Multiprocessamento			x		x
Múltiplas plataformas			x	x	x
Espaço de endereço protegido			x	x	x
GDI		x	(X Windows)	x	x
Capacidade de rede interna			x		x

Tabela 1.1 Uma comparação das características avançadas de vários sistemas operacionais.

Do ponto de vista do programador, o Windows NT elimina a segmentação de memória e os modos de 16 e 32 bits, tem a vantagem de um processamento real de 32 bits, endereços de memória lineares, espaço de endereço protegido, multitarefa preemptiva, e novos recursos gráficos. Além disso, os programas escritos para o Windows NT contam com uma grande variedade de plataformas de hardware, possuindo inclusive uma versão sendo portada para um supercomputador CRAY!

1.4 Referências Bibliográficas

[1] R. S. Wiener, L. J. Pinson, *An Introduction to Object-Oriented Programming and C++*, Addison Wesley, 1988.

[2] M. A. Ellis, B. Stroustrup, *C++ Manual de Referência Comentado*, Campus, 1993.

[3] H. Custer, *Windows NT*, Makron Books do Brasil, 1994.

[4] B. Myers, E. Hamer, *Dominando a Programação no Windows NT*, LTC-Livros Técnicos e Científicos, 1994.

[5] G. Booch, *Object Oriented Design with Applications*, Benjamin/ Cummings, 1991.

[6] P. J. Perry, N. C. Shammas, L. Atkinson, M. Atkinson, *Using Borland C++ 4, Special Edition*, QUE, 1994.

[7] T. Swan, *Programação Avançada em Borland C++ 4.0 para Windows*, Berkeley, 1994.

[8] Microsoft Visual C++ 1.0, *Manuais de Referência*, Microsoft Press, 1994.

[9] B. W. Kernigham, D. M. Richie, *C - A Linguagem de Programação*, Campus, 1986.

Capítulo 2

Estrutura do Programa

2.1 Introdução

Neste capítulo apresentaremos a estrutura do SIMNT, descrevendo cada uma das classes introduzidas no capítulo anterior: *biblioteca*, *blocos*, *parâmetros*, *buffer*, *usuário* e *gerenciador*. Estas classes serão referenciadas por seus respectivos nomes utilizados na implementação do programa em C++ (tabela 2.1).

CLASSE	Nome (C++)	Tipo
biblioteca	BlockLib	base abstrata
blocos	Block	derivada : public BlockLib
parâmetros	BlockParam	derivada : public Block
buffer	BlockOut	derivada : public BlockParam
usuário	BlockUser	derivada : public BlockOut
gerenciador	BlockOrder	derivada : public BlockUser

Tabela 2.1 Classes do SIMNT.

2.2 Classe BlockLib

A estrutura da classe BlockLib é apresentada na figura 2.1, ela é composta de diversos tipos de dados e funções. O vetor *SystemLib* é um vetor de tamanho *MaxLibIndex* que possui todos os parâmetros necessários às classes derivadas, os quais

são armazenados em estruturas do tipo *StructLib*, e são acessados através do índice *LibIndex*: uma variável que indica qual modelo de dispositivo está sendo invocado. A estrutura StructLib é formada por quatro elementos: o nome do modelo *FunctionName*, número de saídas *Nout*, número de entradas *Nin* e um ponteiro *FunctionLib* para a função que implementa o modelo.

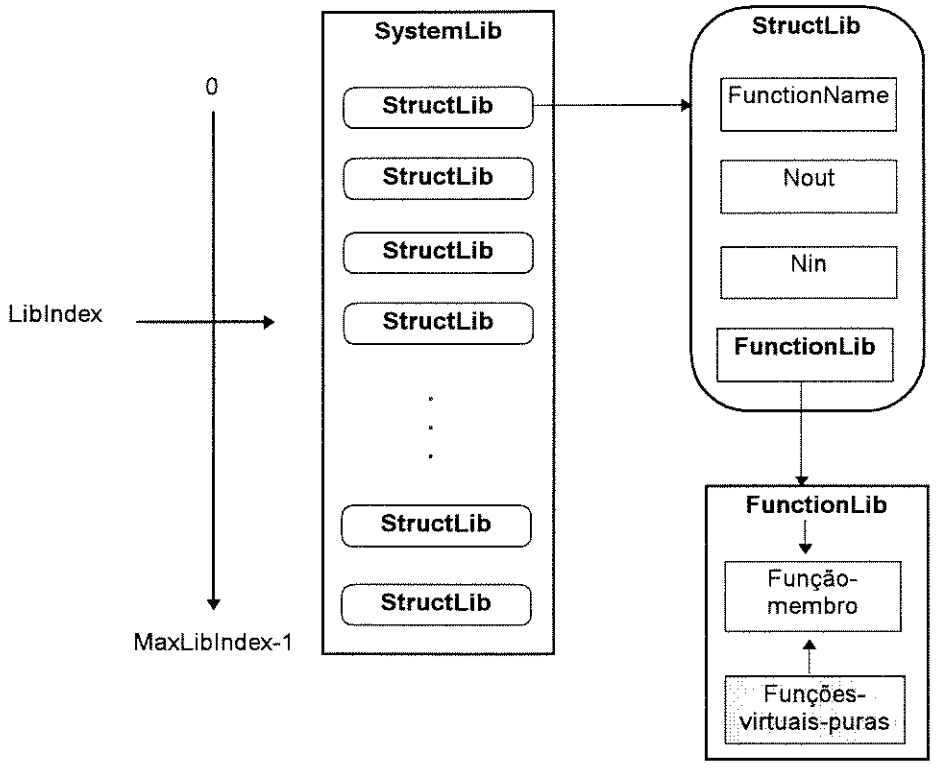


Figura 2.1 Estrutura da classe BlockLib.

As funções-membro que compõe a classe podem ser divididas em três categorias: funções para manipulação de SystemLib, funções associadas aos modelos, e funções-virtuais-puras para implementação do modelos. O primeiro tipo de funções permite a seleção e acesso aos parâmetros do modelo a ser executado. Esta seleção é realizada alterando-se o índice LibIndex. Deste modo, todas as demais funções desta categoria se referem a este índice. Funções criadas para a inclusão de novos modelos irão também fazer parte deste grupo. A segunda categoria é formada por funções que representam os

dispositivos, e são acessadas de maneira indireta por FunctionLib. A terceira categoria corresponde a funções-virtuais-puras. Este tipo de função torna a classe BlockLib uma *classe abstrata*[1], podendo ser unicamente usada como uma classe base para outras classes. Neste caso, estas funções utilizam recursos que somente serão fornecidos pela classe derivada BlockUser, que será descrita posteriormente.

Um dos principais recursos do SIMNT é permitir a ampliação da biblioteca de modelos. Isto implica não apenas a inclusão de novas funções-membro, mas também em seu registro dentro de SystemLib. O registro é realizado dinamicamente durante a inicialização da classe BlockLib.

As classes derivadas de BlockLib inicializam a classe através de uma função-membro de nome *LoadLib*, esta chama uma série de outras funções que adicionam todos os modelos ao SIMNT. Na lista 2.1 temos chamadas para a função *AddLib*, que aloca dinamicamente três novas estruturas StructLib em SystemLib, fornecendo o nome, função, número de entradas e saídas de cada modelo.

```
void BlockLib::LoadLib()
{
        if(Flag==0)
                {
                        AddLib("DRIVERNRZ",&BlockLib::DriverNRZ,0,1);
                        AddLib("SMLASER1",&BlockLib::SMLaser1,1,1);
                        AddLib("FIBER1",&BlockLib::OptFiber1,1,1);
                }
        Flag=1;
}
```

Lista 2.1 Funções de registro dos modelos - módulo loadlib.cpp.

Deste modo, o procedimento para a inclusão de um novo modelo é simples, basta adicionar uma nova chamada para a função AddLib dentro de LoadLib, compilar e ligar o módulo *loadlib.cpp* ao programa principal.

2.3 Classe Block

A classe Block possui um *construtor*[1], que é uma função-membro utilizada para construir valores de seu tipo na classe. Se uma classe tem um construtor, cada objeto desta classe será inicializado antes que seja feito o uso de qualquer outro objeto. No nosso caso, o construtor é inicializado com o nome do arquivo de topologia.

A classe Block interpreta o arquivo de topologia fornecido para simulação de um determinado sistema. Esta classe tem sua estrutura apresentada na figura 2.2, sendo formada por dois tipos de dados. O primeiro é um vetor *System*, composto dos blocos do sistema representados por estruturas *StructBlock*. Já o segundo, representado pela linha pontilhada, é um conjunto de elementos necessários para implementação dos recursos de integração e hierarquia de blocos.

A partir do arquivo de topologia, a classe Block preenche as estruturas StructBlock com os seguintes parâmetros: nome do bloco *Name*, arquivo de parâmetros *File*, modelo *Function*, e uma variável de estado *Status*. Além disso, a classe fornece recursos para sua classe derivada *BlockOut*, calculando o número de buffers necessários e seus endereços de leitura e escrita. Estes endereços estão armazenados nos vetores *AdressIn* e *AdressOut* respectivamente. Durante a simulação outras variáveis de estado são necessárias: *PosIn*, *PosOut*, *Max* e *FileOut*. Os dois primeiros, possuem a atual posição no buffer de saída da próxima amostra, para as entradas e saídas dos blocos; os

dois últimos indicam o número máximo de amostras geradas e seu respectivo arquivo de saída.

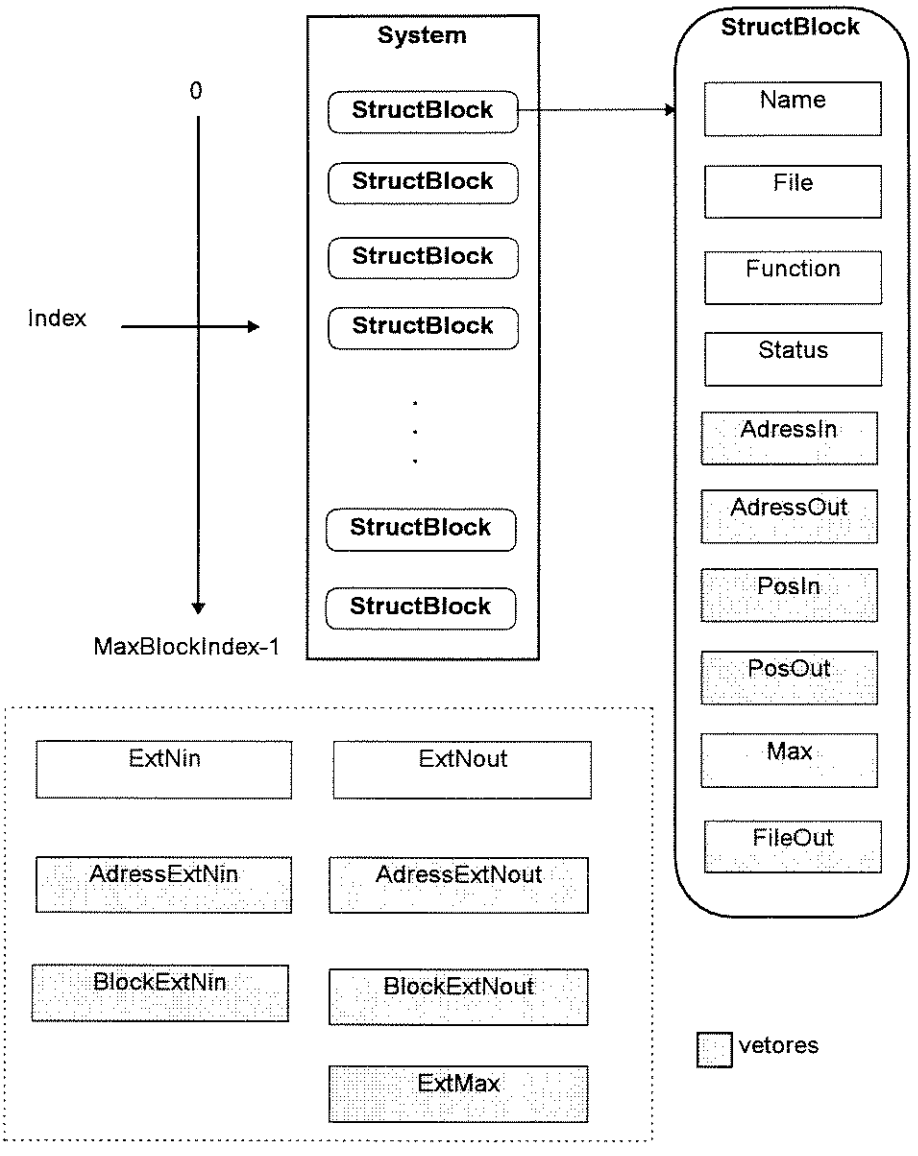


Figura 2.2 Estrutura da classe Block.

Um outro importante recurso do SIMNT é a possibilidade de definirmos um conjunto de blocos como um único bloco, sem necessidade de programação, bastando definir um arquivo texto de topologia do conjunto. Este recurso necessita do uso do

segundo grupo de dados, que informa ao sistema o número de entradas e saídas externas, endereços de leitura e escrita, nome dos blocos e número máximo de amostras, armazenados em *ExtNin*, *ExtNout*, *AdressExtNin*, *AdressExtNout*, *BlockExtNin*, *BlockExtNout* e *ExtMax*, respectivamente.

2.4 Classe BlockParam

A classe BlockParam, mostrada na figura 2.3, constitui-se em um vetor *Param* de estruturas *StructParam*, uma para cada bloco, e é formada por matrizes irregulares de cinco tipos: *Complex*, *Int*, *Double*, *Long*, *Char*, onde seus nomes se referem aos tipos de dados que armazenam. Deste modo, o tipo *complex* é armazenado na matriz *Complex*, com número de linhas *LinComplex*, número de colunas para cada linha em um vetor *ColComplex*, e os nomes destes vetores em um outro vetor *NameComplex*.

Durante a implementação dos modelos, não é possível o uso da palavra reservada *static*[1], pois um modelo pode ter diversas ocorrências num mesmo sistema, entretanto, a classe BlockParam permite a alocação e acesso a variáveis de estado, através de suas funções-membro. Estas variáveis são armazenadas nas matrizes localizadas em StructParam. Este recurso é utilizado durante o processo de simulação.

A classe BlockParam possui um construtor que utiliza os arquivos de parâmetros. É importante destacar que quando nos referirmos a parâmetros dos modelos, estamos nos referindo aos parâmetros específicos dos dispositivos, ou seja, características físicas. O nome destes arquivos são fornecidos pela classe herdada Block. Com estes arquivos, durante a inicialização da classe, o construtor carrega para memória todos os parâmetros de um determinado bloco, que podem ser dos cinco tipos citados, inclusive sob a forma de vetores. Deste modo, quando um bloco solicita algum parâmetro ele não é buscado

diretamente do arquivo, o que tornaria o processo lento, e sim, da classe BlockParam. O ganho de desempenho com este recurso é observado quando um sistema apresenta uma realimentação, que necessita a chamada de um bloco diversas vezes.

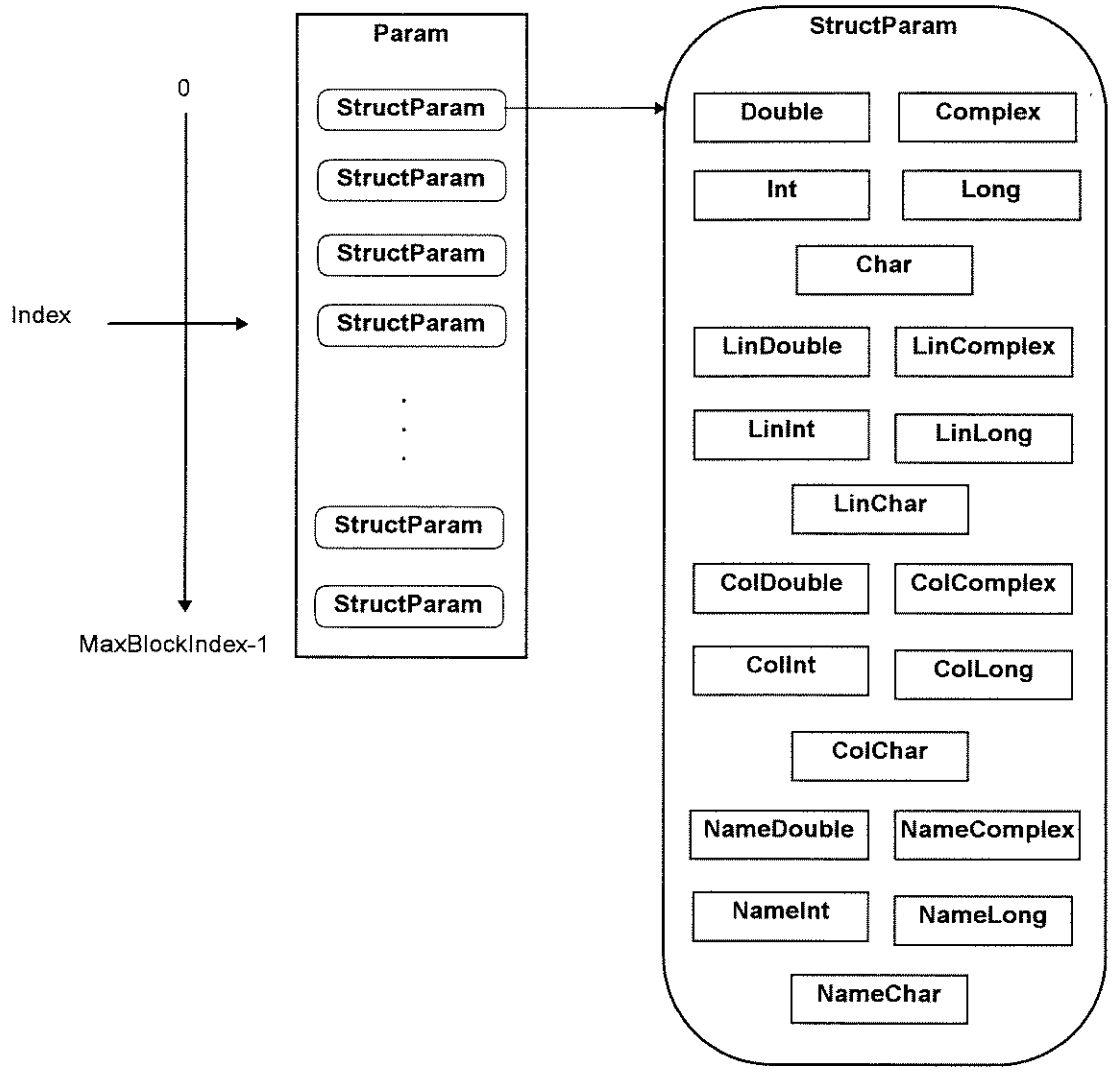


Figura 2.3 Estrutura da classe BlockParam.

2.5 Classe BlockOut

A classe BlockOut é formada por um vetor *SystemOut* de tamanho *MaxCursor* composto de estruturas *StructBlockOut*. MaxCursor é o somatório de todas as saídas de um determinado sistema de blocos. A seleção de uma determinada estrutura *StructBlockOut* é realizada através do índice *Cursor*. Ao final do processo de simulação, a classe BlockOut tem todas as suas estruturas preenchidas.

StructBlockOut é um conjunto de elementos associados a uma saída de um bloco. Estes elementos informam as características das amostras geradas durante a simulação, são elas : um vetor de tipos double *Double*, complex *Complex* ou int *Int*, o tipo do vetor indicado por *Type*, o tamanho do vetor *Size*, o domínio no tempo ou frequência *Domain*, a unidade de medida utilizada *Unit*, a frequência de amostragem das amostras *SampleFreq*, a frequência central (envelope complexo) *CenterFreq*, o tempo da primeira amostra *InitTime*, um nome qualquer para o vetor *Label*, e uma variável de estado *Flag*.

O construtor da classe *BlockOut* aloca o vetor SystemOut com as informações fornecidas pela classe herdada Block e, quando um bloco é chamado durante a execução do programa, ele fornece as amostras geradas e suas características ao escrever na sua saída os resultados. Além disso, a classe BlockOut é responsável pela escrita em arquivo dos resultados (esta opção é feita no arquivo de topologia), onde se indica quais as saídas de interesse para pós-processamento.

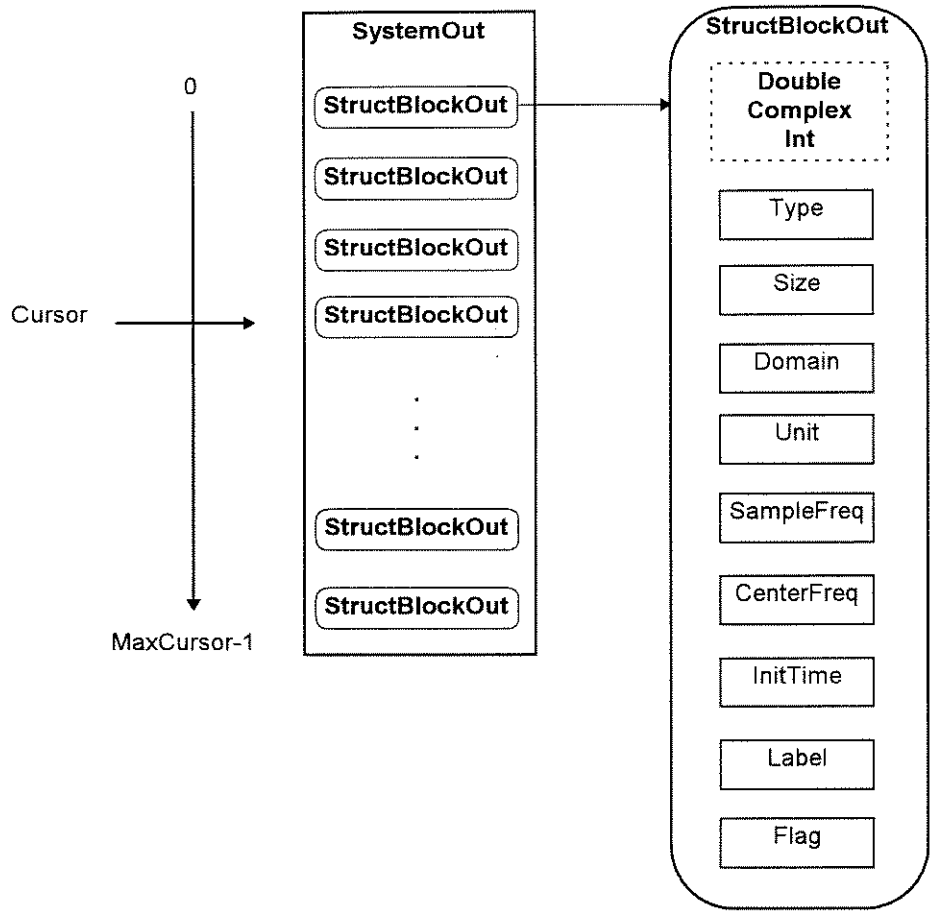


Figura 2.4 Estrutura da classe BlockOut.

2.6 Classe BlockUser

A classe *BlockUser* é formada por mais de cinquenta funções-membro, todas estas funções são *virtuais*[1], utilizadas para o desenvolvimento dos modelos da classe abstrata BlockLib. Estas funções, fornecem ao usuário que pretende desenvolver modelos de novos dispositivos, uma interface padrão para leitura e escrita de dados. A lista 2.2 apresenta algumas destas funções. No apêndice A apresentaremos todas as funções disponíveis para os usuários.

```
1 ReadSize(entrada)
2 ReadMax(saída)
3 ReadDomain(entrada)
4 ReadType(entrada)
5 ReadSampleFreq(entrada)
6 ReadCenterFreq(entrada)
7 ReadInitTime(entrada)
8 WriteStatus(estado)
9 ReadSample(entrada, tamanho)
10 WriteProp(saída,tipo,domínio,unidade, freq. de amost. , freq. central , tempo inicial ,nome)
11 WriteSample(entrada, vetor ,tamanho)
12 READPARAM(parâmetro)
```

Lista 2.2 Algumas funções-membro da classe BlockUser.

Na lista apresentada acima, a função *ReadSize* (linha 1) retorna o número de amostras disponíveis na *entrada* selecionada; *ReadMax* retorna o tamanho máximo do buffer de *saída*; *ReadDomain* retorna o domínio (tempo ou frequência) das amostras; *ReadType* retorna o tipo das amostras (complex, int, double); *ReadCenterFreq*, *ReadSampleFreq* e *ReadInitTime* retornam a frequência central, frequência de amostragem e tempo inicial das amostras, respectivamente; *WriteStatus* informa ao SIMNT qual o estado do bloco (estes estados são quatro: *INIT* - não foi ainda invocado, *DEAD* - já foi invocado e terminou, *DEAD* -deve ser invocado novamente, e finalmente, *STOP* - não pode ser chamado); a função *ReadSample* retorna um vetor com as amostras disponíveis na *entrada,* com um determinado *tamanho*; *WriteProp* escreve na saída todas as propriedades das amostras geradas no bloco; *WriteSample* escreve na saída um vetor de amostras com *tamanho* definido, e retorna um flag de estado informando se o buffer de saída já está cheio; finalmente na linha 12, temos a macro *READPARAM* que retorna na variável *parâmetro* o respectivo valor indicado no arquivo de parâmetros dos dispositivos.

2.7 Classe BlockOrder

Esta classe é responsável pelo gerenciamento e definição da ordem de execução dos blocos, durante a simulação no SIMNT. É formada por um vetor *Schedul* de estruturas *StructOrder*, uma para cada bloco (figura 2.5). Estas estruturas são compostas de um vetor *Way* de tamanho *Size*. Além disso, o vetor *Blocks* possui a ordem da primeira execução do programa, enquanto o vetor *Rep* possui a próxima.

A ordenação e o fluxo de dados é similar ao programa BLOSIM[2], definida a ordenação inicial, cada bloco é chamado, e retorna uma variável de estado que indica se o bloco **terminou** seu processamento, se **deve ser chamado novamente** ou se **não pode ser chamado novamente**. Caso algum dos blocos deva ser chamado novamente, o processo reinicia, com uma nova ordenação em função deste ou destes blocos. A primeira ordenação é realizada no construtor de BlockOrder, e preenche os vetores Blocks e Schedul. Na figura 2.6, temos um sistema full-duplex descrito em[2], utilizaremos este exemplo para demonstrar os critérios adotados para ordenação dos blocos. O processo de gerenciamento da ordenação segue a seguinte rotina:

A) Procuramos por blocos que não possuem entradas (primeiro grupo):

0-FAR TRANMITTER 1-NEAR TRANSMITTER 2-NOISE GENERATOR

B) Procuramos por blocos ligados exclusivamente ao primeiro grupo:

3-CHANNEL FILTER 4-SINK 5-ECHO FILTER 6-CANCELLER

C) Blocos ligados exclusivamente ao primeiro e segundo grupo:

7-ECHO DELAY

D) Blocos ligados exclusivamente ao primeiro, segundo e terceiro grupo:

8-CHANNEL ADDER

E) Blocos ligados exclusivamente aos grupos anteriores:

9-ANTI-ALIAS FILTER

F) E assim sucessivamente:

10-RECEIVER ADDER 11-FORK 12-TIMING 13-INTERPOLATION FILTER

14-RECEIVER 15-ERROR MONITOR

Duas observações cabem aqui, quando alguns dos grupos ficam vazios, o critério passa a ser a procura de blocos com pelo menos uma ligação com o grupo anterior, e ao final do processo, blocos que não se encaixam em algum dos grupos são incluídos no final da lista. Deste modo, temos o vetor Blocks preenchido com a primeira ordem de execução.

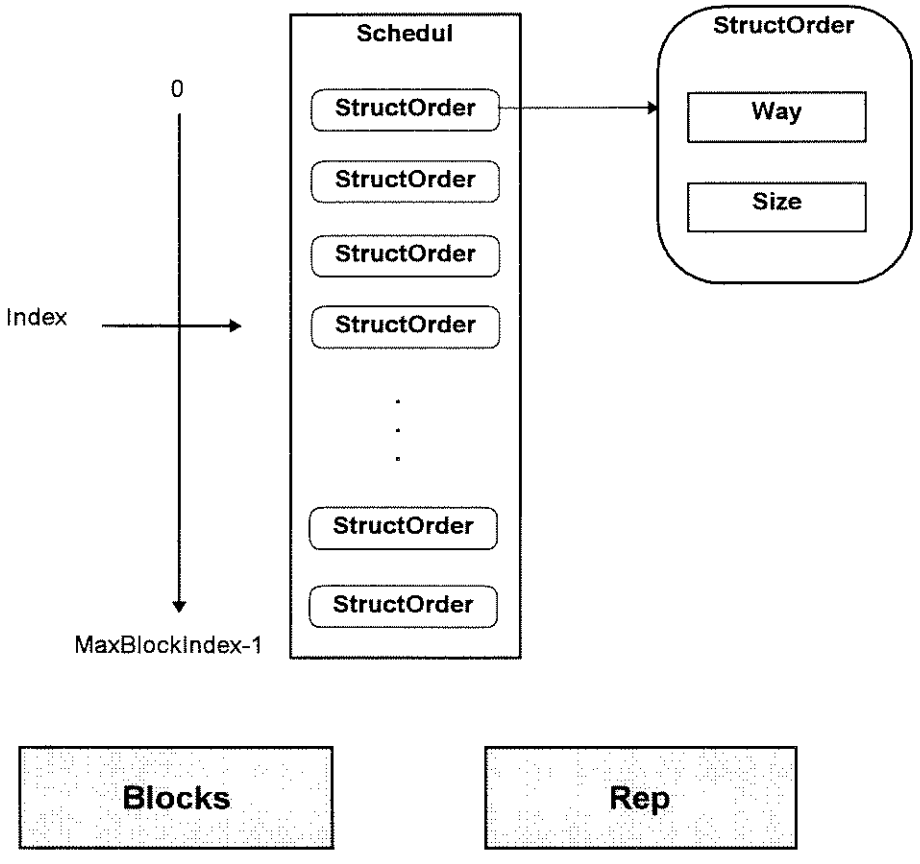


Figura 2.5 Estrutura da classe BlockOrder.

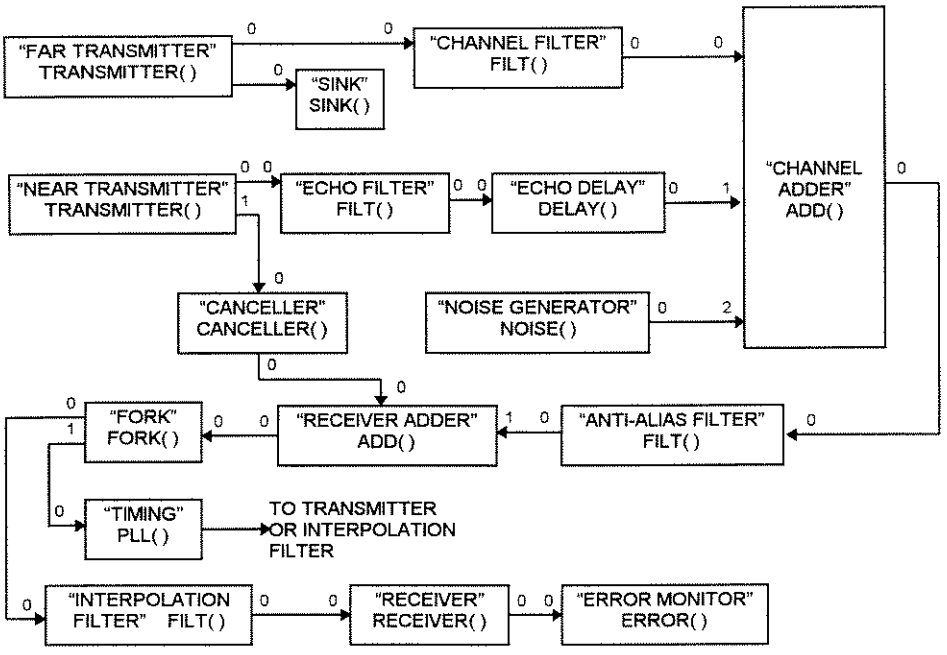


Figura 2.6 Diagrama de blocos de um sistema full-duplex de transmissão de dados[2].

Os vetores Way também devem ser preenchidos. Neste caso, a diferença básica no critério de ordenação, é que o primeiro grupo é composto de apenas um determinado bloco, e os grupos seguintes são formados sem a necessidade de estarem ligados exclusivamente com os grupos anteriores. Este procedimento fornece um aumento de velocidade no processos de reordenação durante a simulação no SIMNT.

A execução dos blocos inicialmente é realizada na ordem definida no vetor Blocks. Caso um ou mais blocos devam ser chamados novamente, seus vetores *Way* são adicionados, ordenados e colocados no vetor Rep, que define a nova ordem de execução; por exemplo, na figura 2.6 vamos supor que apenas os blocos *NEAR TRANSMITTER* e *SINK* devam ser chamados novamente, assim teremos que o conteúdo do vetor Rep será: *0-NEAR TRANSMITTER 1-ECHO FILTER 2-CANCELLER 3-ECHO DELAY 4-CHANNEL ADDER 5-ANTI-ALIAS FILTER 6-RECEIVER ADDER 7-FORK 8-TIMING 9-INTERPOLATION FILTER 10-RECEIVER 11-ERROR MONITOR.*

A simulação termina quando todos os blocos não requisitarem nova chamada, ou quando os buffers de saída estiverem cheios. Este procedimento é idêntico para subsistemas, ou seja, quando algum bloco indique um grupo de outros blocos, eles serão ordenados obedecendo o mesmo critério.

O recurso de multienlace[3] do Windows NT permite que um grupo de blocos seja executado simultaneamente, o que aumenta em muito a velocidade de execução de um determinado sistema. Este recurso será implementado em uma versão futura do SIMNT.

2.8 Referências Bibliográficas

[1] M. A. Ellis, B. Stroustrup, *C++ Manual de Referência Comentado*, Campus, 1993.

[2] D. G. Messerschmitt, "A Tool for Structured Functional Simulation", *IEEE Journal of Selected Areas in Communications*, SAC-2, No. 1, pp 137-147, janeiro 1984.

[3] H. Custer, *Windows NT*, Makron Books do Brasil, 1994.

Capítulo 3

Funcionamento do SIMNT

3.1 Introdução

Neste capítulo apresentamos a estrutura de funcionamento do SIMNT.

A figura 3.1 mostra o diagrama do fluxo de operações no programa. Cada bloco se refere a dados e funções pertencentes a uma ou mais classes, enquanto as ligações representam o caminho onde transitam dados e mensagens entre estes elementos.

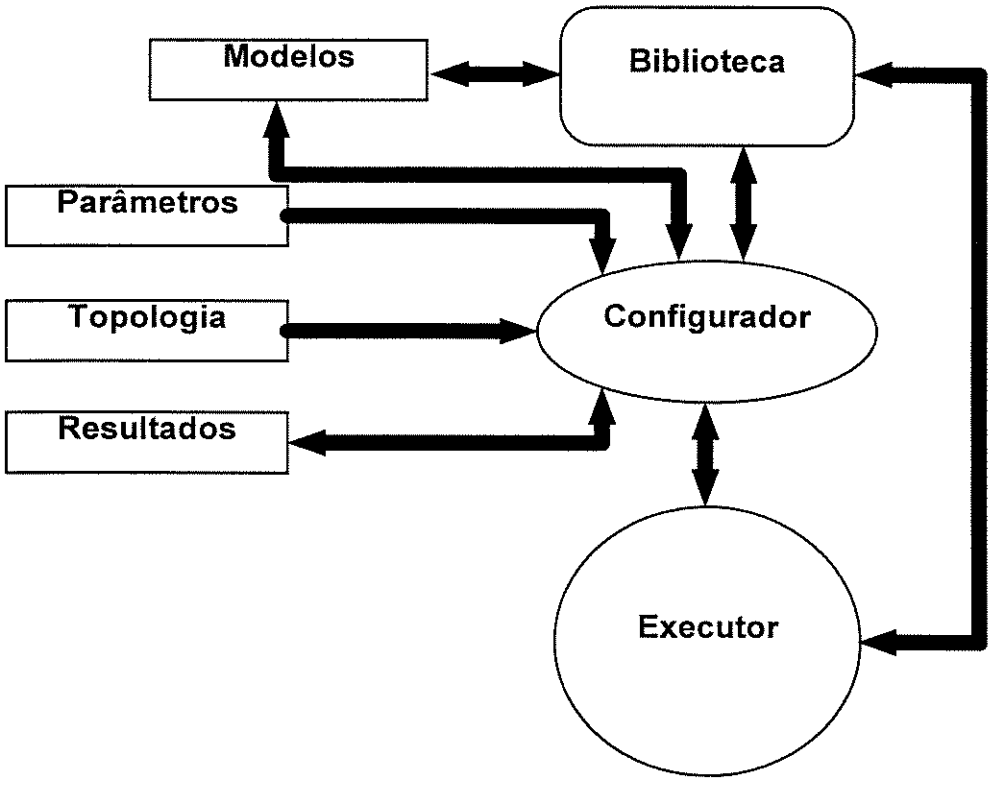


Figura 3.1 Fluxo de operações no SIMNT.

O *configurador* é formado pelos construtores das classes Block, BlockParam, BlockOut e BlockOrder, associados com os arquivos de *topologia, parâmetros* e *resultados.* Deste modo, armazena todas as variáveis de estado relacionadas com cada um dos blocos de um determinado sistema. A biblioteca é formada pelo construtor da classe *BlockLib,* e os *modelos* são funções-membro desta classe. Finalmente, o *executor* é responsável pela chamada dos construtores das classes e o inicio da simulação.

O executor invoca os modelos através da biblioteca, e estes utilizam variáveis de estados fornecidas pelo configurador.

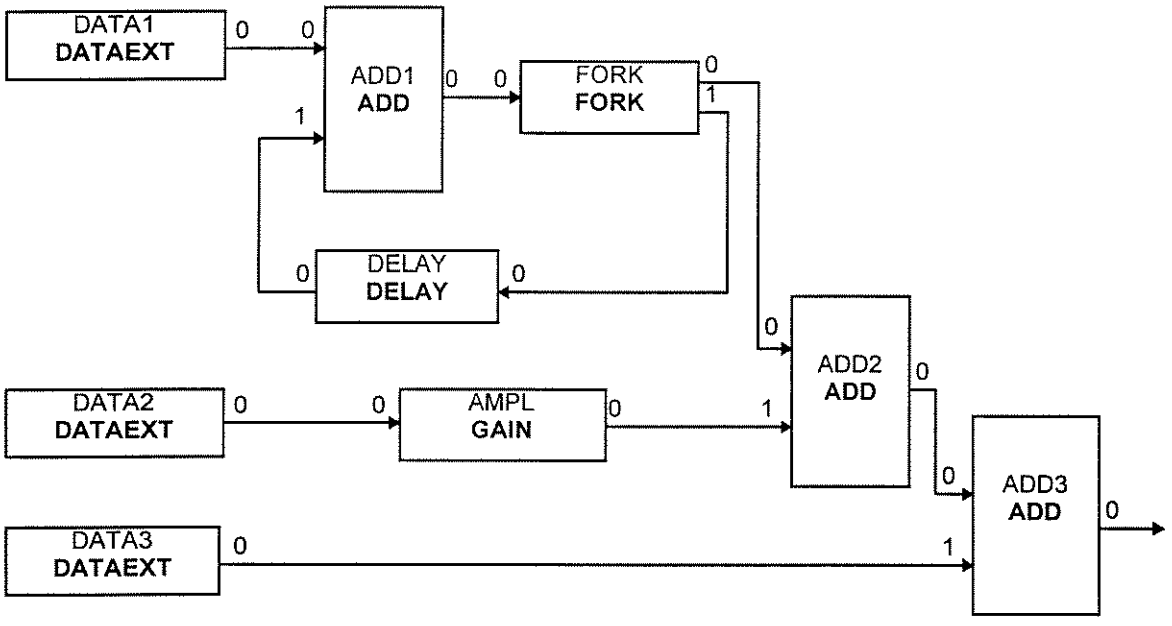


Figura 3.2 Sistema de blocos.

Neste capítulo apresentaremos o funcionamento e o fluxo de operações do SIMNT através da simulação de um sistema. Esta simulação será descrita passo a passo, desde a programação dos modelos até a geração dos resultados finais. O sistema proposto encontra-se na figura 3.2, sendo formado por nove blocos: *DATA1*, *DATA2*, *DATA3*, *ADD1*, *ADD2*, *ADD3*, *AMPL*, *DELAY* e *FORK*. Os três primeiros são ocorrências do modelo *DATAEXT*, os três seguintes, ocorrências do modelo *ADD*, e

finalmente, AMPL, DELAY e FORK, ocorrências dos modelos *GAIN*, *DELAY*, *FORK* respectivamente.

Este sistema foi implementado por razões didáticas, pois permite demonstrar o funcionamento e todos os recursos do SIMNT. No próximo capítulo apresentaremos a simulação de um sistema de comunicação óptica real.

3.2 Implementação dos Modelos

A etapa inicial da simulação de um sistema é o modelamento dos dispositivos. Entretanto, quando a biblioteca fornece um número de modelos suficientes, não é necessário seguirmos este procedimento. Nesta seção apresentaremos as características e a programação dos cinco dispositivos da figura 3.2.

3.2.1 DATAEXT

Este dispositivo carrega do seu arquivo de parâmetros uma sequência de amostras, e coloca estas amostras na sua única saída. Os parâmetros fornecidos são: frequência central *SampleFreq*, frequência de amostragem *CenterFreq*, tempo inicial *InitTime*, sequência de amostras *seq*, e seu tamanho *size*. O programa em C++ de implementação do modelo é fornecido na lista 3.1, onde as funções fornecidas ao usuário estão destacadas em negrito.

```
#include "blocklib.h"
#include "blockord.h"

void BlockLib::DataExt()
{

        long size;
        READPARAM(size);
        double SampleFreq;
        READPARAM(SampleFreq);
        double CenterFreq;
        READPARAM(CenterFreq);
        double InitTime;
        READPARAM(InitTime);
        double *seq=new double [size];
        READPARAM(seq);

        WriteProp(0,DOUBLE,TIME,0,SampleFreq,CenterFreq,InitTime,"DataExt");
        if(WriteSample(0,seq,size)==OVERF)
                                {
                                        WriteStatus(STOP);
                                }
        else WriteStatus(DEAD);
        delete[] seq;
}
```

Lista 3.1 Listagem do arquivo DATAEXT.CPP.

Inicialmente é declarada a função-membro da classe BlockLib *void DataExt(void).* Cada um dos parâmetros é declarado com seu respectivo tipo e nome, antes da chamada a macro *READPARAM.* No caso de vetores, devemos antes alocar a memória necessária para o mesmo, como podemos observar para o parâmetro *seq.*

A seguir a função *WriteProp* registra na saída zero o tipo, domínio, e as demais características das amostras. Finalmente, uma chamada à função *WriteSample* coloca na saída zero o vetor *seq,* de tamanho *size,* retornando um flag de estado. Caso seja *OVERF,* indicando que o buffer de saída já está cheio, o bloco informa ao *executor* que não pode ser chamado novamente, enviando na função *WriteStatus* o estado *STOP,* senão envia *DEAD,* indicando que terminou seu processamento, mas pode ser chamado novamente.

3.2.2 ADD

Este dispositivo realiza a soma das amostras disponíveis nas suas duas entradas e coloca o resultado na sua saída, como podemos observar na lista 3.2.

```
#include "blocklib.h"
#include "blockord.h"

void BlockLib::Add()
{
  double  *xd0;
  double  *xd1;
  double  *yd0;

  long i=0;
  long size,size1,size2;

  size1=ReadSize(0);
  size2=ReadSize(1);

  WriteProp(0,DOUBLE,ReadDomain(0),0,ReadSampleFreq(0),ReadCenterFreq(0),ReadInitTime(0),"Add");
  if((size1==0)||(size2==0)) WriteStatus(DEAD);
   else
    {
      WriteStatus(AGAIN);
      if(size1==size2)
        {
          size=size1;
          WriteStatus(DEAD);
        }
      else if(size1>size2) size=size2;
      else size=size1;

      xd0= (double  *) ReadSample(0,size);

      xd1= (double  *) ReadSample(1,size);

      yd0 = new double[size];

      for(i=0;i<size;i++) yd0[i]=xd0[i]+xd1[i];
      if(WriteSample(0,yd0,size)==OVERF)
        {
          WriteStatus(STOP);
        }

      delete[] yd0;
      delete[] xd0;
      delete[] xd1;
    }
}
```

Lista 3.2 Listagem do arquivo ADD.CPP.

Este somador foi desenvolvido para somar vetores de amostras de qualquer tipo, *DOUBLE, COMPLEX* ou *INT*, entretanto, na lista 3.2 apresentamos somente uma

versão simplificada do mesmo. As funções *ReadSize* retornam o número de amostras disponíveis nas entradas do somador nas variáveis *size1* e *size2,* respectivamente. Quando não há amostras disponíveis nas duas entradas size1 e size2 são zero, e a função *WriteStatus* informa ao executor que o bloco terminou (*DEAD*). Caso as duas sejam diferentes de zero, mas de valor igual, é realizada a soma das amostras, neste caso o bloco também informa que terminou (*DEAD*). Finalmente, quando size1 e size2 são de tamanhos diferentes, o estado do bloco é *AGAIN*, além disso, o número de amostras geradas é o menor valor entre size1 e size2, ou seja, *size*.

As duas chamadas à função *ReadSample,* retornam dois vetores de tamanho *size* com as amostras das entradas 0 e 1, respectivamente em *xd0* e *xd1*. O resultado da soma é colocado em *yd0*.

3.2.3 DELAY

Este dispositivo tem um único parâmetro, um valor em segundos *delay.* A função *WriteDelay,* na lista 3.3, introduz um atraso de *del* amostras na saída zero deste bloco. Devemos observar que ele determina o tipo do vetor de amostras na entrada zero, para definir o tipo para saída, isto pode ser observado na função reservada **switch**. Este procedimento deve ser adotado para otimização da memória, caso contrário bastaríamos definir todos os tipos utilizados no SIMNT como COMPLEX.

3.2.4 GAIN

Este dispositivo multiplica as amostras de sua entrada zero pelo parâmetro *Ganho,* verificando os tipos de amostras conforme podemos observar na lista 3.4.

```
#include "blocklib.h"
#include "blockord.h"

void BlockLib::Delay()
{

  double delay;
  READPARAM(delay);

  double  *xd0;
  complex  *xc0;
  int  *xi0;
  long del= delay*ReadSampleFreq(0);
  WriteDelay(0,del);

  int type=ReadType(0);
  long size=ReadSize(0);
  WriteProp(0,type,ReadDomain(0),0,ReadSampleFreq(0),ReadCenterFreq(0),ReadInitTime(0),"Delay");
  if(size==0) WriteStatus(DEAD);
  else
    {
      switch(type)
        {
          case DOUBLE:
            {
              xd0= (double  *) ReadSample(0,size);
              if(WriteSample(0,xd0,size)==OVERF)
                {
                  WriteStatus(STOP);
                }
              else WriteStatus(DEAD);
              delete[] xd0;
              break;
            }
          case COMPLEX:
            {
              xc0= (complex  *) ReadSample(0,size);
              if(WriteSample(0,xc0,size)==OVERF)
                {
                  WriteStatus(STOP);
                }
              else WriteStatus(DEAD);
              delete[] xc0;
              break;
            }
          case INT:
            {
              xi0= (int  *) ReadSample(0,size);
              if(WriteSample(0,xi0,size)==OVERF)
                {
                  WriteStatus(STOP);
                }
              else WriteStatus(DEAD);
              delete[] xi0;
              break;
            }
        }
    }
}
```

Lista 3.3 Listagem do arquivo DELAY.CPP.

```
#include "blocklib.h"
#include "blockord.h"

void BlockLib::Gain()
{
  double Ganho;
  READPARAM(Ganho);
  double  * xd0;
  complex  * xc0;
  int  * xi0;
  int type=ReadType(0);
  long size=ReadSize(0);
  WriteProp(0,type,ReadDomain(0),0,ReadSampleFreq(0),ReadCenterFreq(0),ReadInitTime(0),"Ampl");
  if(size==0) WriteStatus(DEAD);
  else
    {
      switch(type)
        {
          case DOUBLE:
            {
              xd0= (double  *) ReadSample(0,size);
              for(long i=0;i<size;i++) xd0[i]=Ganho*xd0[i];

              if(WriteSample(0,xd0,size)==OVERF)
                  {
                    WriteStatus(STOP);
                  }
              else WriteStatus(DEAD);
              delete[] xd0;
              break;
            }
          case COMPLEX:
            {
              xc0= (complex  *) ReadSample(0,size);
              for(long i=0;i<size;i++) xc0[i]=Ganho*xc0[i];

              if(WriteSample(0,xc0,size)==OVERF)
                  {
                    WriteStatus(STOP);
                  }
              else WriteStatus(DEAD);
              delete[] xc0;
              break;
            }
          case INT:
            {
              xi0= (int  *) ReadSample(0,size);
              for(long i=0;i<size;i++) xi0[i]=Ganho*xi0[i];

              if(WriteSample(0,xi0,size)==OVERF)
                  {
                    WriteStatus(STOP);
                  }
              else WriteStatus(DEAD);
              delete[] xi0;
              break;
            }
        }
    }
}
```

Lista 3.4 Listagem do arquivo GAIN.CPP.

```
#include "blocklib.h"
#include "blockord.h"

void BlockLib::Fork()
{
  double   * xd0;
 complex   * xc0;
 int   * xi0;
 int type=ReadType(0);
 long size=ReadSize(0);
 WriteProp(0,type,ReadDomain(0),0,ReadSampleFreq(0),ReadCenterFreq(0),ReadInitTime(0),"Fork");
 WriteProp(1,type,ReadDomain(0),0,ReadSampleFreq(0),ReadCenterFreq(0),ReadInitTime(0),"Fork");
 if(size==0) WriteStatus(DEAD);
 else
  {
   switch(type)
    {
     case DOUBLE:
      {
       xd0= (double   *) ReadSample(0,size);
       if((WriteSample(0,xd0,size)==OVERF)||(WriteSample(1,xd0,size)==OVERF))
        {
         WriteStatus(STOP);
        }
       else WriteStatus(DEAD);
       delete[] xd0;
       break;
      }
     case COMPLEX:
      {
       xc0= (complex   *) ReadSample(0,size);
       if((WriteSample(0,xc0,size)==OVERF)||(WriteSample(1,xc0,size)==OVERF))
        {
         WriteStatus(STOP);
        }
       else WriteStatus(DEAD);
       delete[] xc0;
       break;
      }
     case INT:
      {
       xi0= (int   *) ReadSample(0,size);
       if((WriteSample(0,xi0,size)==OVERF)||(WriteSample(1,xi0,size)==OVERF))
        {
         WriteStatus(STOP);
        }
       else WriteStatus(DEAD);
       delete[] xi0;
       break;
      }
    }
  }
}
```

Lista 3.5 Listagem do arquivo FORK.CPP.

3.2.5 FORK

FORK é utilizado para bifurcações, onde uma saída deve ser duplicada. No nosso caso, isto é necessário, porque a saída de ADD1 é conectada à entrada de DELAY

e ADD2. Sua implementação é fornecida na lista 3.5. Aqui observamos que ocorrem duas chamadas às funções *WriteProp* e *WriteSample,* uma para cada saída.

3.3 Arquivos de Topologia e Parâmetros

Para simularmos um determinado sistema no SIMNT, devemos obedecer regras de formato para os arquivos que definem a topologia do sistema e seus parâmetros. Para o sistema da figura 3.2, temos o arquivo de topologia apresentado na lista 3.6. As linhas iniciadas com o caractere "{" definem o nome do bloco, modelo e, quando houver, o arquivo de parâmetros. O caractere "[" informa a ligação entre a saída de um bloco com a entrada de outro, além disso, o tamanho máximo do buffer de saída também é fornecido. O caractere "<" informa a saída de interesse para pós-processamento, definindo o nome, saída e o respectivo arquivo. Finalmente, o caractere "#" permite a inclusão de comentários no arquivo.

Nenhum bloco pode ter entradas em aberto, entretanto, é permitido que suas saídas não estejam conectadas a nenhum bloco. No caso de uma saída ficar em aberto, o SIMNT ignora esta saída, não reservando um buffer para as amostras geradas. Para a saída do bloco ADD3 não ficar em aberto, conectamos ao nome reservado *OUT*, que informa ao SIMNT que este bloco necessita de um buffer de saída.

```
# SISTEMA DA FIGURA 3.2
# NOME          MODELO                  ARQUIVO DE PARÂMETROS

{ DATA1         DATAEXT                     DATA1.PAR
{ DATA2         DATAEXT                     DATA2.PAR
{ DATA3         DATAEXT                     DATA3.PAR
{ DELAY         DELAY                       DELAY.PAR
{ AMPL          GAIN                        AMPL.PAR
{ FORK          FORK
{ ADD1          ADD
{ ADD2          ADD
{ ADD3          ADD

# NOME          SAÍDA   NOME        ENTRADA     TAMANHO

[ DATA1         0       ADD1        0           256
[ DELAY         0       ADD1        1           256
[ ADD1          0       FORK        0           256
[ FORK          0       ADD2        0           256
[ FORK          1       DELAY       0           256
[ DATA2         0       AMPL        0           256
[ AMPL          0       ADD2        1           256
[ ADD2          0       ADD3        0           256
[ DATA3         0       ADD3        1           256
[ ADD3          0       OUT                     256

# NOME          SAÍDA   ARQUIVO DE SAÍDA

< DATA1         0       DATA1_0.DAT
< DELAY         0       DELAY_0.DAT
< ADD1          0       ADD1_0.DAT
< DATA2         0       DATA2_0.DAT
< AMPL          0       AMPL_0.DAT
< ADD2          0       ADD2_0.DAT
< DATA3         0       DATA3_0.DAT
< ADD3          0       ADD3_0.DAT
```

Lista 3.6 Listagem do arquivo SISTEMA1.TOP.

O arquivo de parâmetros deve informar o tipo, o nome e o valor do parâmetro. São admitidos os seguintes tipos : *double*, *int*, *complex*, *char* e *long*, inclusive sob forma de vetores. Na lista 3.7, observamos parâmetros do tipo *double* e *long*, onde *seq* é um vetor do tipo *double*. Esta é uma listagem parcial, pois este arquivo possui mais de 250 linhas em função do parâmetro *seq*. Um exemplo mais completo é fornecido na lista 3.8. Aqui temos os seguintes parâmetros:

- Um vetor do tipo double: x[0]=1.89, x[1]=4.90, x[2]=2.088, x[3]=5.998, x[4]=4.998, x[5]=8.990.
- Um vetor de strings: palavras[0]="codigo1", palavras[1]="cod2", palavras[2]="cod3", palavras[4]="cod4".
- Um valor do tipo int : y=2.
- Um vetor do tipo int: z[0]=7, z[1]=8, z[2]=9.

- Um vetor do tipo complex: k[0]=(1.2,4.5), k[1]=(3.45,7.980).
- Um vetor do tipo long: i[0]=900, i[1]=987, i[2]=9987, i[3]=8987, i[4]=7868.i[5]=12235, i[6]=55643, i[7]=3456.

```
# parâmetros para DATA

double InitTime 0
double SampleFreq 79.629E9
double CenterFreq 1.9608E9
long size 256
double seq
0.03885
0.03885
0.03885
0.03885
0.03885
0.03885
0.03885
0.03885
```

Lista 3.7 Listagem parcial do arquivos DATA1.PAR, DATA2.PAR e DATA3.PAR.

```
# EXEMPLO DE ARQUIVO DE PARÂMETROS

double x 1.89 4.90 2.088 5.998
4.998 8.990
char palavras codigo1 cod2 cod3
cod4
int y 2
int z 7 8 9
complex k (1.2,4.5) (3.45,7.980)
long i 900 987 9987 8987
7868
12235
55643
3456
```

Lista 3.8 Exemplo dos formatos para os parâmetros.

A lista 3.9 e a lista 3.10 são os arquivos de parâmetros para o bloco AMPL e DELAY, respectivamente. Os demais arquivos não possuem parâmetros.

```
# Arquivo de parâmetros para AMPL

double Ganho 10
```

Lista 3.9 Listagem do arquivo AMPL.PAR

```
# Arquivo de parâmetros para DELAY
double delay 3.5E-10
```

Lista 3.10 Listagem do arquivo DELAY.PAR.

3.4 Descrição do Funcionamento do SIMNT

Literalmente, o SIMNT é um objeto da classe *BlockOrder*, com um único parâmetro inicial, o arquivo de topologia. Com este arquivo o processo de configuração das classes é iniciado. Vamos descrever as operações de inicialização e configuração do SIMNT, utilizando o diagrama da figura 3.3, aplicado ao nosso exemplo (figura 3.2), o que necessariamente envolve a descrição das operações durante a chamada dos construtores das classes:

1) Construtor da classe *Block* :

1.1) Chamada para função *LoadLib*, que informa à classe *BlockLib* (biblioteca de modelos) os modelos disponíveis para simulação.

1.2) Preenchimento de *SystemLib*, um vetor de estruturas *StructLib*. Cada uma destas estruturas é associada a um dos modelos e armazena suas características (nome e número do modelo, número de entradas e saídas, etc). Seu estado interno é mostrado na tabela 3.1.

1.3) Leitura do arquivo *SISTEMA1.TOP*, determinando o número de buffers de escrita para o sistema, além disso, para cada bloco é definido o endereço de escrita e leitura para suas saídas e entradas nestes buffers. Nas tabelas 3.2-3.6 apresentamos o estado interno inicial das estruturas *StructBlock*, uma para cada bloco,

responsáveis pelo armazenamento destas características no vetor *System*. A descrição simplificada destas características se encontra na segunda coluna das tabelas 3.2-3.6.

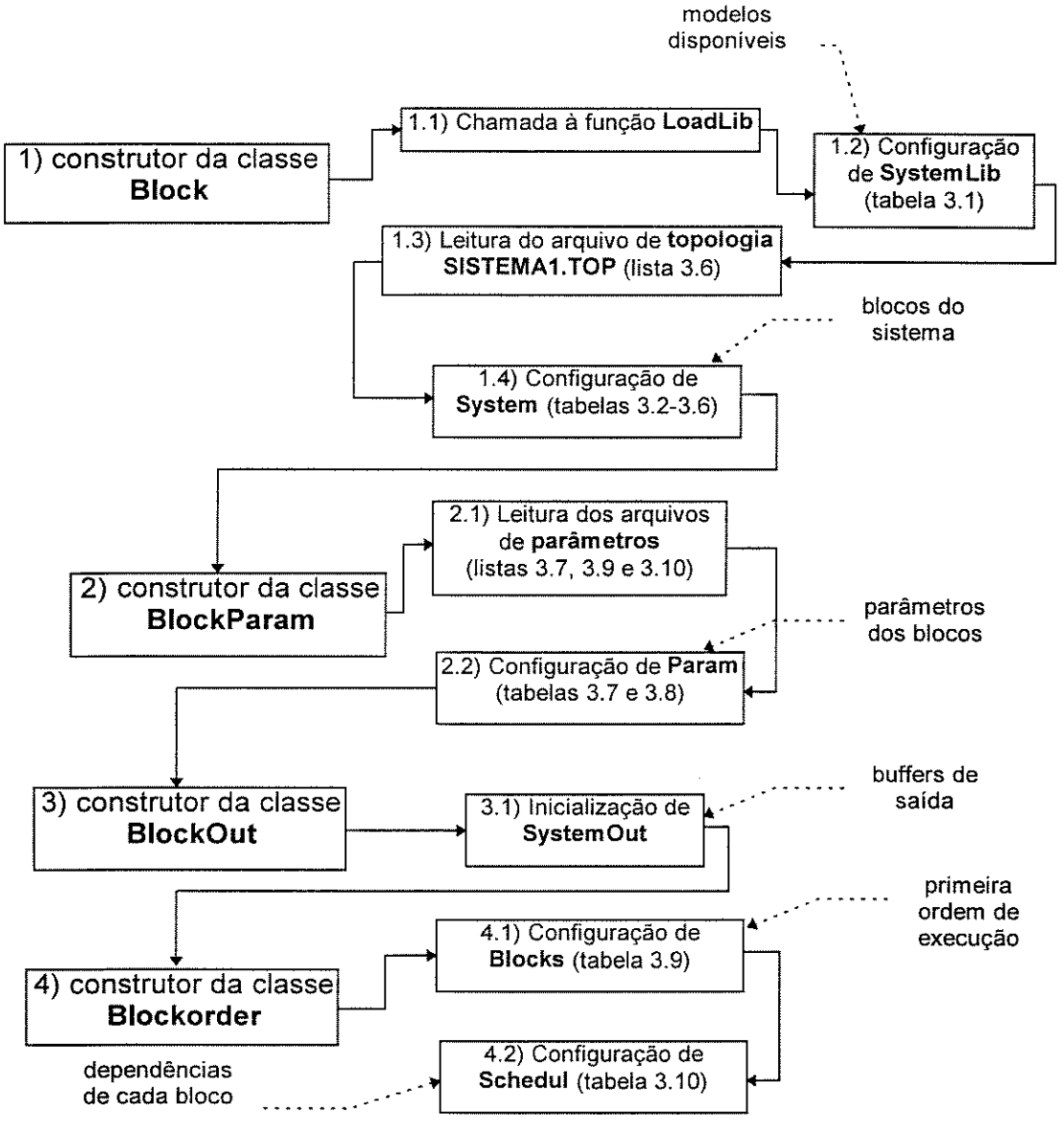


Figura 3.3 Fluxo de operações durante a configuração do SIMNT.

Variável	Descrição					
LibIndex	*número do modelo*	0	1	2	3	4
FunctionName	*nome do modelo*	DATAEXT	ADD	DELAY	FORK	GAIN
Nin	*número de entradas*	0	2	1	1	1
Nout	*número de saídas*	1	1	1	2	1
FunctionLib	*função-membro da classe*	DataExt	Add	Delay	Fork	Gain

Tabela 3.1 Estado interno das estruturas *StructLib* em *SystemLib*.

Variável	Descrição		
Index	*número do bloco*	0	
Name	*nome do bloco*	DATA1	
File	*arquivo de parâmetros*	DATA1.PAR	
Function	*nome do modelo*	DATAEXT	
Status	*estado do bloco*	LIVE	
AdressIn	*endereços de leitura nos buffers de saída*	-	
Posin	*posição de leitura nos buffers de saída*	-	
AdressOut	*endereços de escrita nos buffers de saída*	saída 0	0
PosOut	*posição de escrita nos buffers de saída*	saída 0	0
Max	*número máximo de amostras geradas*	saída 0	256
FileOut	*arquivos de saída*	saída 0	DATA1_0.DAT

Tabela 3.2 Estado interno da estrutura *StructBlock* para o bloco DATA1.

Variável			
Index	1		2
Name	DATA2		DATA3
File	DATA2.PAR		DATA3.PAR
Function	DATAEXT		DATAEXT
Status	LIVE		LIVE
AdressIn	-		-
Posin	-		-
AdressOut	saída 0	1	2
PosOut	saída0	0	0
Max	saída 0	256	256
FileOut	saída 0	DATA2_0.DAT	DATA3_0.DAT

Tabela 3.3 Estado interno das estruturas *StructBlock* para os blocos DATA2 e DATA3.

Variável			
Index	3		4
Name	DELAY		AMPL
File	DELAY.PAR		AMPL.PAR
Function	DELAY		GAIN
Status	LIVE		LIVE
AdressIn	entrada 0	6	1
Posin	entrada 0	0	0
AdressOut	saída 0	3	4
PosOut	saída 0	0	0
Max	saída 0	256	256
FileOut	saída 0	DELAY_0.DAT	AMPL_0.DAT

Tabela 3.4 Estado interno das estruturas *StructBlock* para os blocos DELAY e AMPL.

Variável		
Index	5	
Name	FORK	
File	-	
Function	FORK	
Status	LIVE	
AdressIn	entrada 0	7
Posin	entrada 0	0
AdressOut	saída 0 saída 1	5 6
PosOut	saída 0 saída 1	0 0
Max	saída 0 saída 1	256 256
FileOut	saída 0 saída 1	- -

Tabela 3.5 Estado interno da estrutura *StructBlock* para o bloco FORK.

Variável				
Index	6		7	8
Name	ADD1		ADD2	ADD3
File	-		-	-
Function	ADD		ADD	ADD
Status	LIVE		LIVE	LIVE
AdressIn	entrada 0	0	5	8
	entrada 1	3	4	2
Posin	entrada 0	0	0	0
	entrada 1	0	0	0
AdressOut	saída 0	7	8	9
PosOut	saída 0	0	0	0
Max	saída 0	256	256	256
FileOut	saída 0	ADD1_0.DAT	ADD2_0.DAT	ADD3_0.DAT

Tabela 3.6 Estado interno das estruturas *StructBlock* para os blocos ADD1, ADD2 e ADD3.

2) Construtor da classe *BlockParam* :

2.1 e 2.2) A partir dos nomes dos arquivos fornecidos pela classe herdada *Block*, os parâmetros são carregados para memória nas estruturas *StructParam* no vetor *Param*. As tabelas 3.7 e 3.8 mostram o estado interno das estruturas StructParam, onde observamos que os parâmetros mostrados nos arquivos das listas 3.7, 3.9 e 3.10 (DATA1.PAR, DATA2.PAR, DATA3.PAR, DELAY.PAR e AMPL.PAR), encontram-se armazenados nestas estruturas.

Variável	Descrição	DELAY	AMPL
LinDouble	*número de parâmetros double*	1	1
NameDouble	*nomes dos parâmetros*	delay	Ganho
ColDouble	*tamanho de cada vetor da matriz*	1	1
Double	*matriz irregular de tipos double*	10	3.5e-10

Tabela 3.7 Estado interno das estruturas da classe *BlockParam* para os parâmetros de DELAY e AMPL.

Variável	DATA1	DATA2	DATA3
LinDouble	4	4	4
NameDouble	InitTime SampleFreq CenterFreq seq	InitTime SampleFreq CenterFreq seq	InitTime SampleFreq CenterFreq seq
ColDouble	1 1 1 256	1 1 1 256	1 1 1 256
Double	0 79.629e9 1.9608e9 0.0385 0.0385 0.0385...	0 79.629e9 1.9608e9 0.0385 0.0385 0.0385...	0 79.629e9 1.9608e9 0.0385 0.0385 0.0385...
LinLong	1	1	1
NameLong	size	size	size
ColLong	1	1	1
Long	256	256	256

Tabela 3.8 Estado interno das estruturas da classe *BlockParam* para os parâmetros de DATA1, DATA2 e DATA3.

3) Construtor da classe *BlockOut* :

3.1) Aloca o número necessário de estruturas *StructBlockOut* (buffer de escrita e características das amostras) em *SystemOut* (9 estruturas no nosso exemplo), mas só preenche as mesmas durante a chamada dos blocos, pois a simulação ainda se encontra no estado de configuração. Apenas quando os blocos forem executados é que teremos amostras em suas saídas, devidamente caracterizadas pela frequência de amostragem, frequência central, tempo inicial etc.

4) Construtor da classe *BlockOrder* :

4.1) Inicialmente realiza a ordenação de todos os blocos e preenche a estrutura *Blocks*, conforme a tabela 3.9. Esta estrutura é utilizada para a primeira execução dos blocos, e posteriormente como referência para reordenações.

4.2) A seguir, preenche cada uma das estruturas *StructOrder* em *Schedul*, conforme a tabela 3.10. Cada estrutura armazena as ligações de um bloco (vetor *Way*) a

partir de suas saídas com o restante do sistema. Esta é a última etapa na fase de configuração.

Variável	**Descrição**	
Blocks	*vetor com a primeira ordem de execução dos blocos*	DATA1 DATA2 DATA3 AMPL ADD1 FORK ADD2 ADD3 DELAY

Tabela 3.9 Estado do vetor *Blocks*.

Nome	Vetor de ordenação do bloco - *Way*	**Tamanho** de *Way*
DATA1	DATA1 ADD1 FORK ADD2 ADD3 DELAY	6
DATA2	DATA2 AMPL ADD2 ADD3	4
DATA3	DATA3 ADD3	2
AMPL	AMPL ADD2 ADD3	3
ADD1	ADD1 FORK ADD2 ADD3 DELAY	5
FORK	FORK DELAY ADD1 ADD2 ADD3	5
ADD2	ADD2 ADD3	2
ADD3	ADD3	1
DELAY	DELAY ADD1 FORK ADD2 ADD3	5

Tabela 3.10 Estado da estruturas em *BlockOrder*.

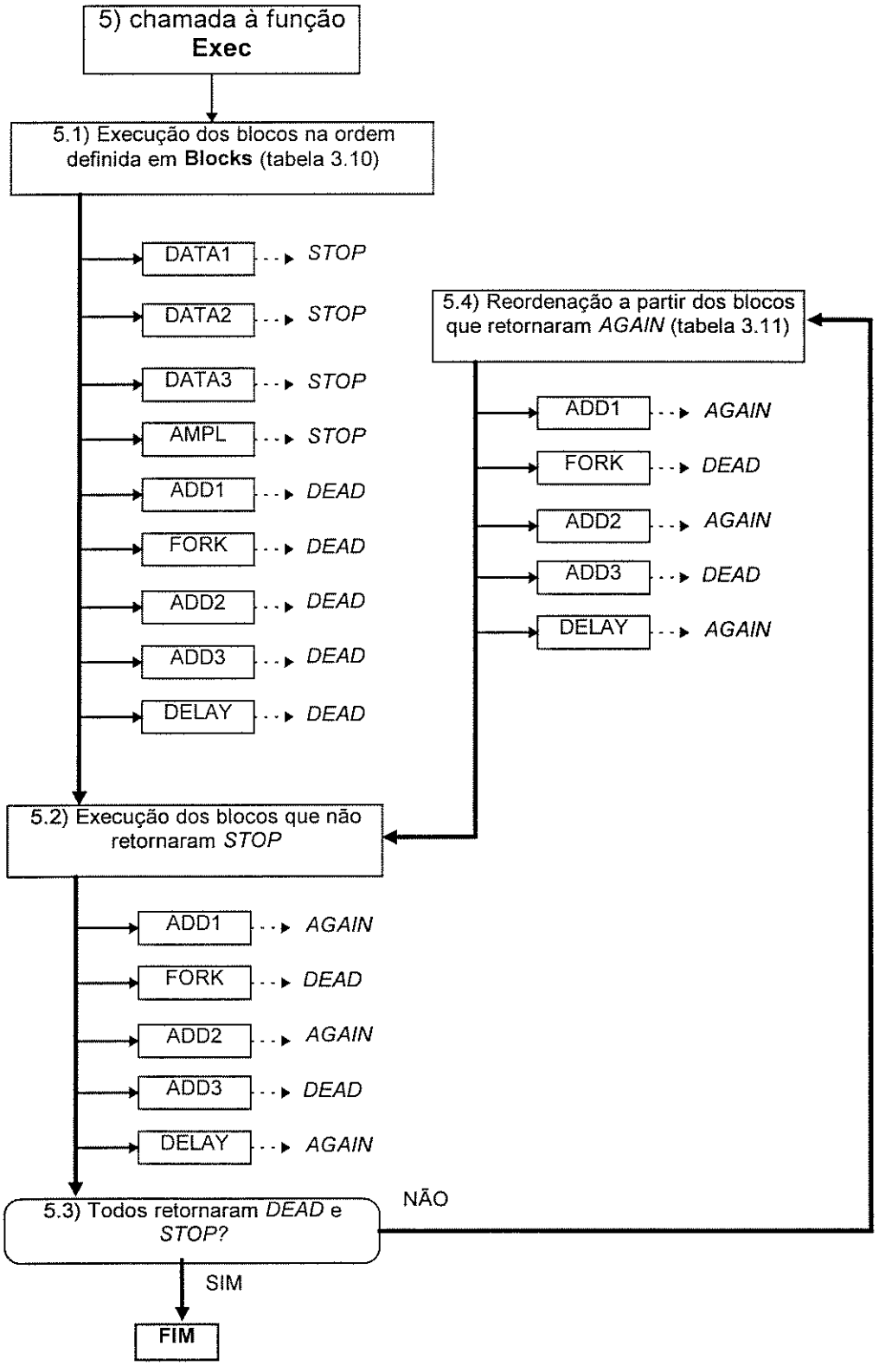


Figura 3.4 Diagrama do fluxo de operações durante a execução dos blocos.

5) O diagrama do fluxo de operações na fase de simulação, ou seja, execução dos blocos, é representada na figura 3.4,e inicia com a chamada à função-membro *Exec*.

5.1) A primeira execução dos blocos segue a ordem definida em Blocks (tabela 3.9):

- DATA1: gera 256 amostras na saída zero. retorna *STOP* (buffer cheio).
- DATA2: gera 256 amostras na saída zero, retorna *STOP* (buffer cheio).
- DATA3: gera 256 amostras na saída zero, retorna *STOP* (buffer cheio).
- AMPL: carrega 256 amostras na entrada zero e gera 256 amostras na saída zero, retorna *STOP* (buffer cheio).
- ADD1: entrada um vazia, retorna *DEAD.*
- FORK: entradas vazias, retorna *DEAD.*
- ADD2: entrada zero vazia, retorna *DEAD.*
- ADD3: entrada zero vazia, retorna *DEAD.*
- DELAY: entrada zero vazia, gera uma amostra (valor zero) na saída zero, retorna *DEAD.*

5.2) Blocos que retornam STOP, estão com os buffers de saída cheios e não dispõe de espaço para armazenar as amostras geradas em uma próxima chamada, estes blocos não serão chamados novamente. Como nenhum bloco retornou AGAIN, serão executados os blocos que retornaram DEAD:

- ADD1: entrada zero com uma amostra (gerada por DELAY) e entrada 1 com 256 amostras, gera uma amostra na saída zero, retorna *AGAIN* (deve ser chamado novamente).
- FORK: entrada zero com uma amostra, coloca esta amostra nas saídas zero e um, retorna DEAD.
- ADD2: entrada zero com uma amostra (gerada por FORK) e entrada 1 com 256 amostras, gera uma amostra na saída zero, retorna *AGAIN* (deve ser chamado novamente).
- ADD3: entrada zero com uma amostra (gerada por ADD2) e entrada 1 com 256 amostras, gera uma amostra na saída zero, retorna *AGAIN* (deve ser chamado novamente).
- DELAY: entrada zero com uma amostra, gera uma amostra na saída zero, retorna *DEAD.*

5.3) Como o DELAY gerou uma amostra na primeira chamada, os somadores dispõem de amostras em suas duas entradas, e retornam *AGAIN.* Caso os blocos tivessem retornado somente *DEAD* e *STOP* a simulação terminaria.

5.4) Devido à realimentação introduzida por *DELAY*, todo o processo passa a ser realizado com uma amostra de cada vez, sendo necessário a chamada aos blocos diversas vezes, isto é feito automaticamente por *Exec.* A próxima chamada aos blocos é armazenada em *Rep*, conforme a tabela 3.11. Este processo de chamada em *Rep* é realizado até que todos os blocos retornem *DEAD* e *STOP*, ainda assim, serão

chamados os que não retornaram *STOP*, se depois desta etapa retornarem apenas *DEAD* e *STOP* a simulação termina[1].

Variável	Descrição	
Rep	*vetor com a próxima ordem de execução*	ADD1 FORK ADD2 ADD3 DELAY

Tabela 3.11 Ordem de execução dos blocos restantes.

Finalmente são gerados os arquivos de saída para pós-processamento, estas amostras estão armazenadas nas estruturas *StructBlockOut* em *SystemOut* na classe *BlockOut*, conforme a tabela 3.12 e 3.13.

Variável	Descrição				
Cursor	*índice do buffer*	0	1	2	3
Type	*tipo do vetor*	DOUBLE	DOUBLE	DOUBLE	DOUBLE
Double Complex Int	*vetor de amostras, apenas um dos três*				
Size	*tamanho do vetor*	256	256	256	256
Domain	*domínio - tempo ou frequência*	TIME	TIME	TIME	TIME
Unit	*unidade das amostras - 0 :nenhuma*	0	0	0	0
SampleFreq	*frequência de amostragem*	7.9629e+10	7.9629e+10	7.9629e+10	7.9629e+10
CenterFreq	*frequência central*	1.9608e+09	1.9608e+09	1.9608e+09	1.9608e+09
InitTime	*tempo inicial*	0	0	0	0
Label	*informação sobre o buffer*	DataExt	DataExt	DataExt	Delay

Tabela 3.12 Estado interno das estruturas da classe BlockOut para os buffers 0,1, 2 e 3.

[1] Se um grupo de blocos retorna DEAD e STOP em duas tentativas, indica que não há mais amostras para processamento (DEAD) ou os buffers estão cheios (STOP).

Variável						
Cursor	4	5	6	7	8	9
Type	DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE
Double Complex Int						
Size	256	256	256	256	256	256
Domain	TIME	TIME	TIME	TIME	TIME	TIME
Unit	0	0	0	0		
SampleFreq	7.9629e+10	7.9629e+10	7.9629e+10	7.9629e+10	7.9629e+10	7.9629e+10
CenterFreq	1.9608e+09	1.9608e+09	1.9608e+09	1.9608e+09	1.9608e+09	1.9608e+09
InitTime	0	0	0	0	0	0
Label	Ampl-Gain K	Fork - 0	Fork - 1	Add	Add	Add

Tabela 3.13 Estado interno das estruturas da classe BlockOut para os buffers 4, 5, 6, 7, 8 e 9.

Nas figuras a seguir apresentamos os resultados gerados ao final da simulação, podemos observar que o número máximo de amostras geradas foi de 256 elementos, entretanto, poderíamos ter especificado um valor superior, limitado apenas pela memória do sistema. A amplitude dos sinais é adimensional. As figuras 3.5-3.7 mostram as saídas dos blocos DATA, que introduzem no sistema as 256 amostras dos seus arquivos de parâmetros. A figura 3.8 é a saída do bloco amplificador, conforme o esperado, esta figura é equivalente a amplitude da figura 3.6 multiplicada pelo ganho 10. As figuras restantes (3.9-3.12) mostram um crescimento da amplitude do sinal na saída dos blocos, característica típica de sistemas instáveis.

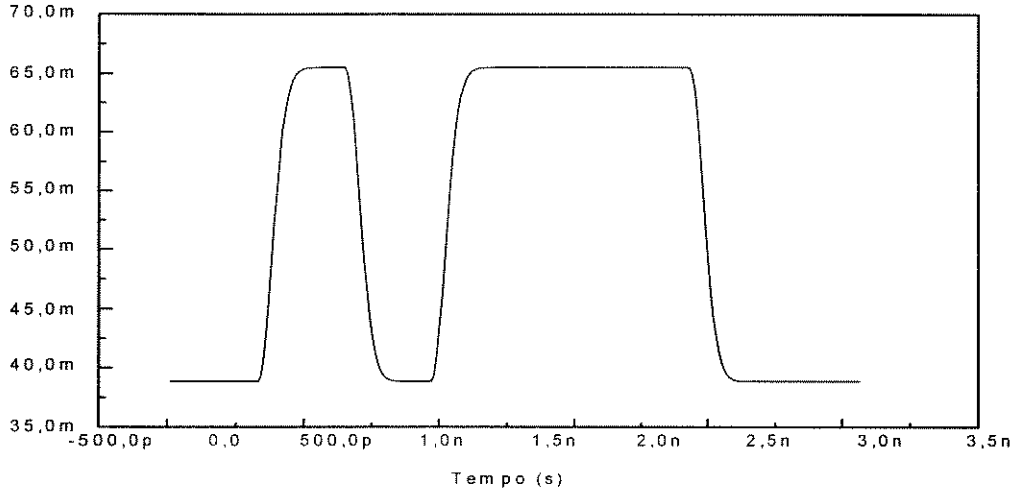


Figura 3.5 Plotagem do arquivo DATA1_0.DAT, saída zero de DATA1.

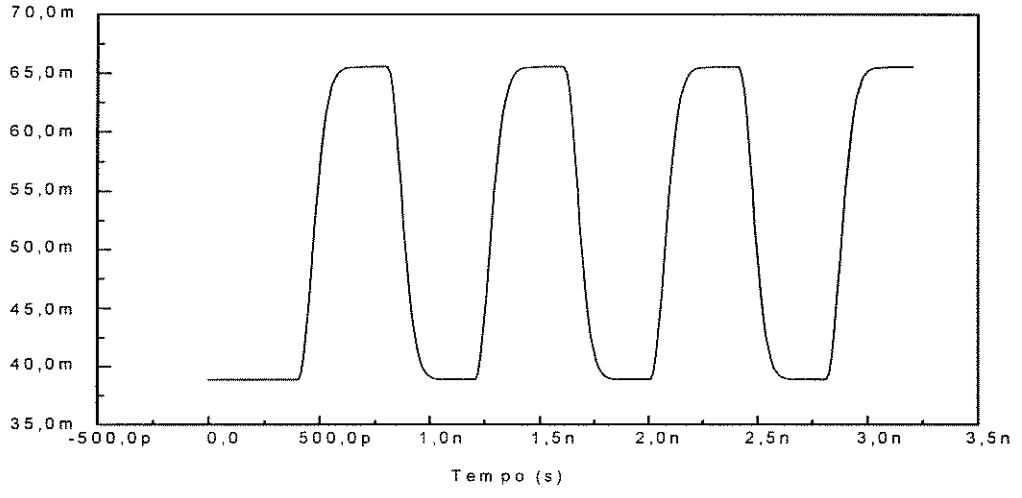


Figura 3.6 Plotagem do arquivo DATA2_0.DAT, saída zero de DATA2.

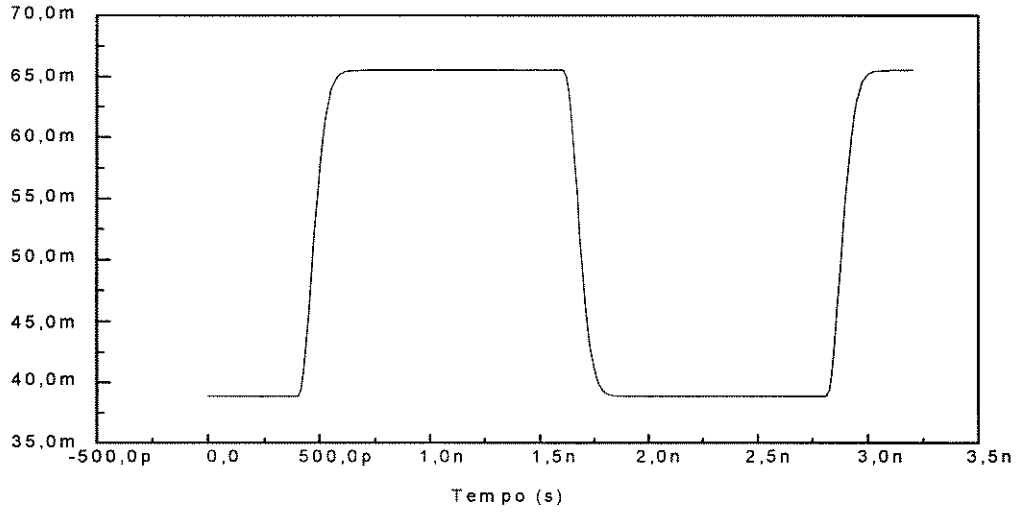


Figura 3.7 Plotagem do arquivo DATA3_0.DAT, saída zero de DATA3.

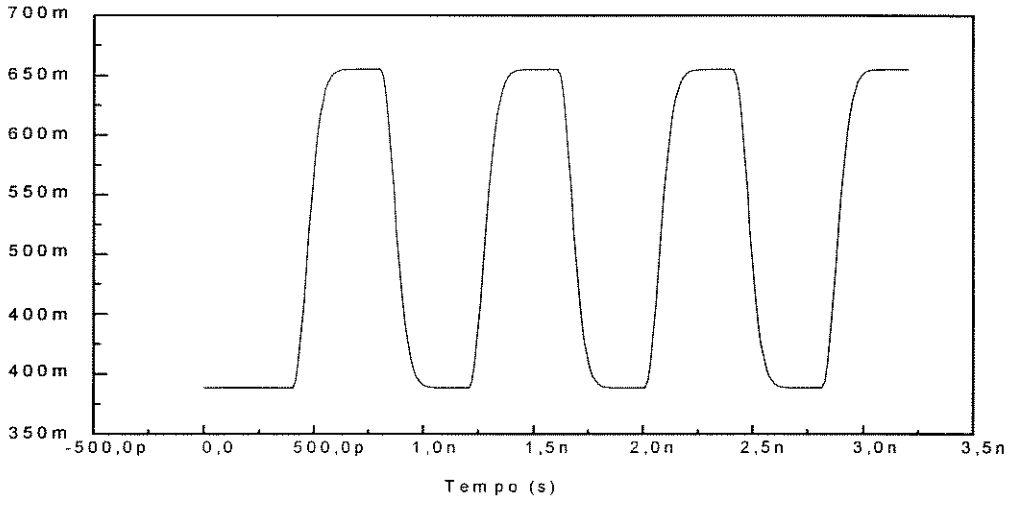


Figura 3.8 Plotagem do arquivo AMPL_0.DAT, saída zero de AMPL.

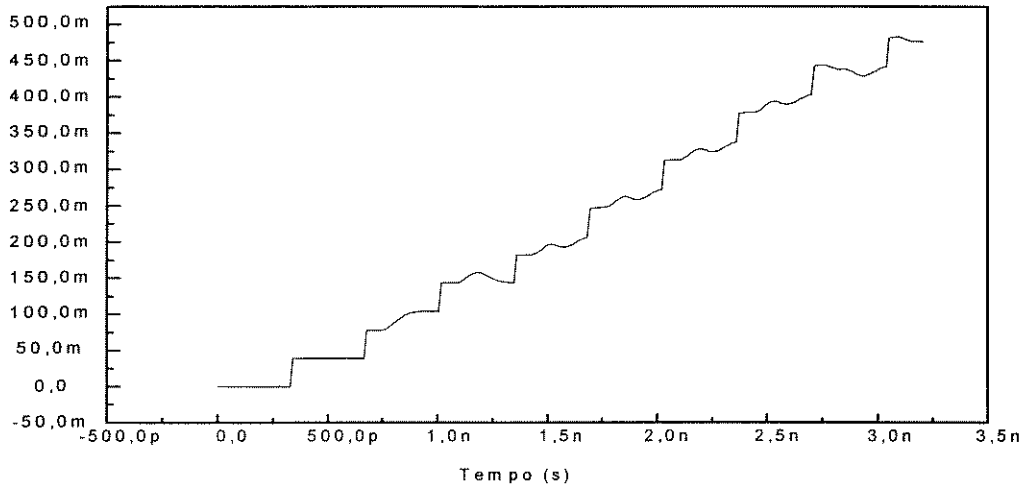


Figura 3.9 Plotagem do arquivo DELAY_0.DAT, saída zero de DELAY.

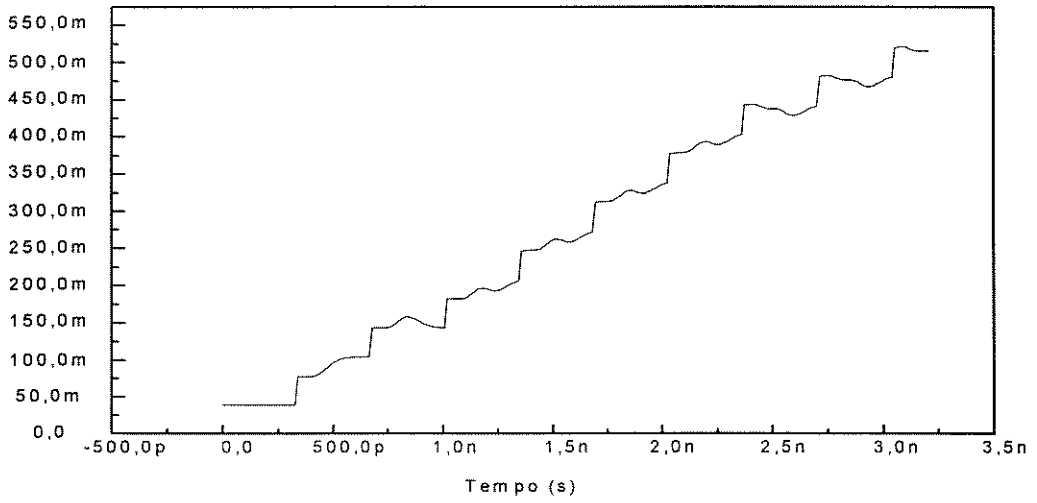


Figura 3.10 Plotagem do arquivo ADD1_0.DAT, saída zero de ADD1.

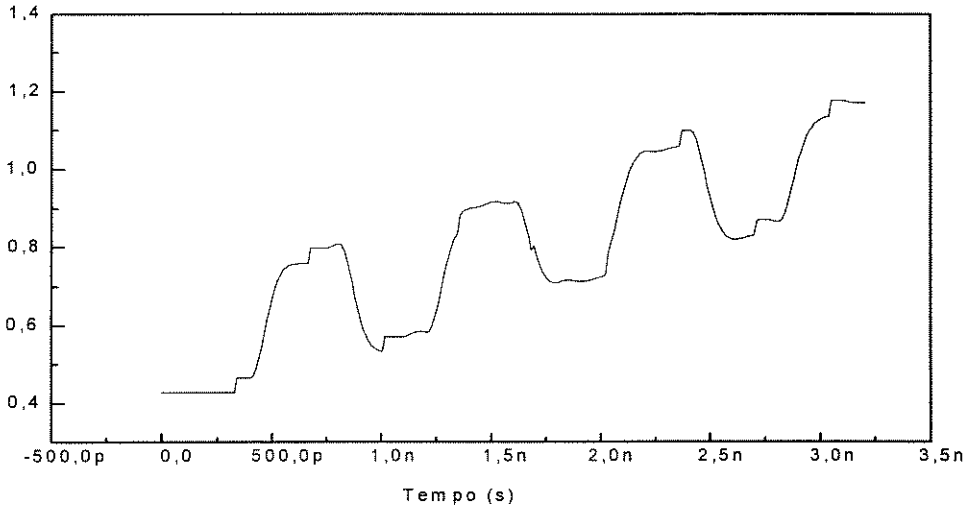


Figura 3.11 Plotagem do arquivo ADD2_0.DAT, saída zero de ADD2.

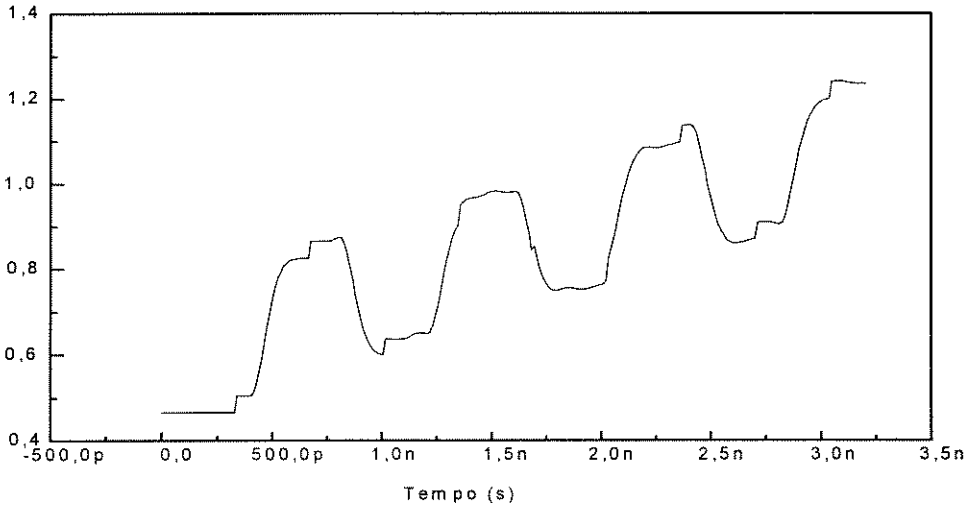


Figura 3.12 Plotagem do arquivo ADD3_0.DAT, saída zero deADD3.

3.5 Estruturando Blocos

O SIMNT permite uma estrutura hierárquica dos blocos, por exemplo, na figura 3.13 temos o mesmo sistema da figura 3.2, onde LOOP e ADD representam um grupo de blocos, conforme a figura 3.14 e figura 3.15, respectivamente. Nós podemos definir um sistema complexo como um grupo de subsistemas, além disso, estes subsistemas podem ser formados por outros, e assim sucessivamente.

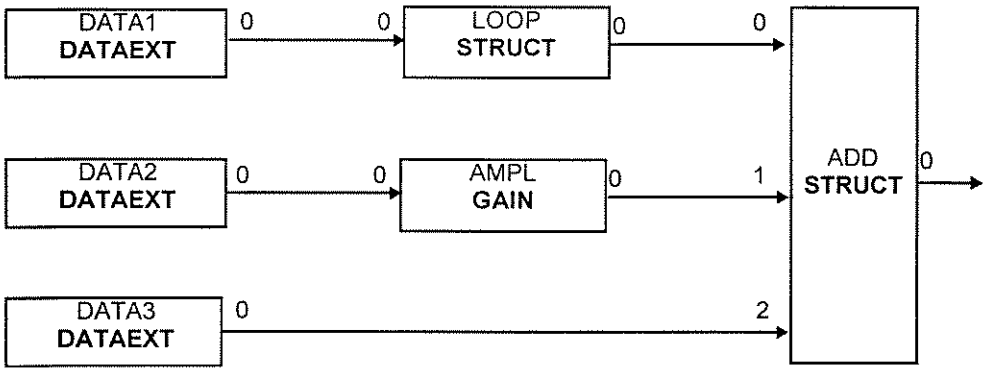


Figura 3.13 Sistema de blocos otimizado.

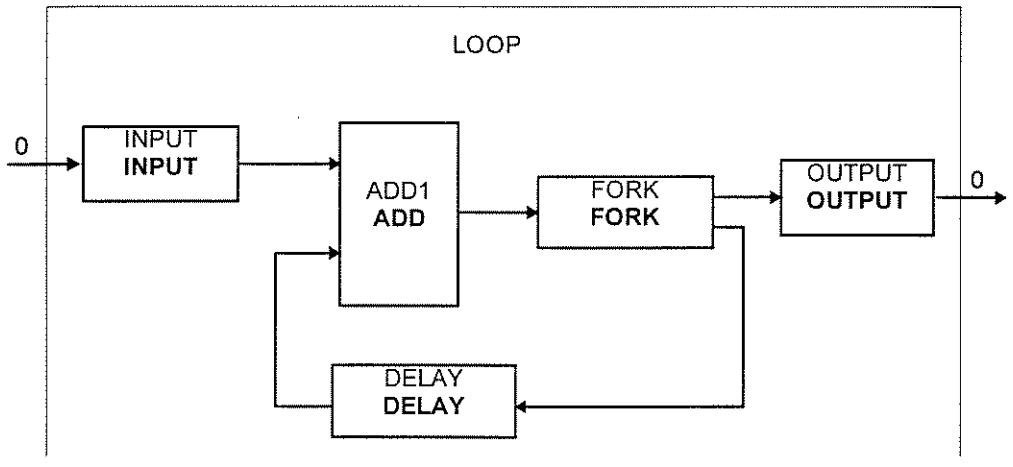


Figura 3.14 Subsistema LOOP.

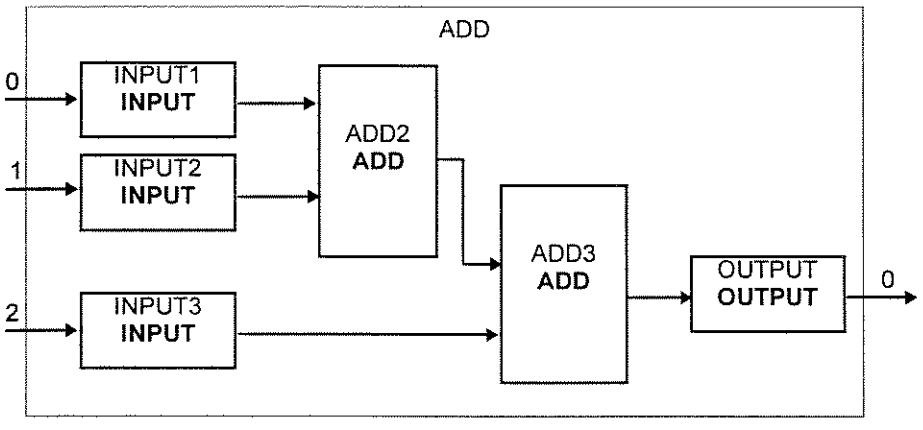


Figura 3.15 Subsistema ADD com três entradas.

```
{ DATA1  DATAEXT   DATA1.PAR
{ DATA2  DATAEXT   DATA2.PAR
{ DATA3  DATAEXT   DATA3.PAR
{ LOOP   STRUCT    LOOP.TOP
{ AMPL   GAIN      AMPL.PAR
{ ADD    STRUCT    ADD.TOP

[ DATA1  0  LOOP   0  256
[ LOOP   0  ADD    0  256
[ DATA2  0  AMPL   0  256
[ AMPL   0  ADD    1  256
[ DATA3  0  ADD    2  256
[ ADD    0  OUT       256

< DATA1  0  DATA1S_0.DAT
< LOOP   0  LOOPS_0.DAT
< ADD    0  ADDS_0.DAT
< DATA2  0  DATA2S_0.DAT
< AMPL   0  AMPLS_0.DAT
< DATA3  0  DATA3S_0.DAT
```

Lista 3.11 Arquivo de topologia SISTEMA2.TOP, para o sistema da figura 3.13.

Os arquivos de topologia utilizam três tipos de blocos para criação de subsistemas: STRUCT, INPUT e OUTPUT. O primeiro recebe um arquivo de topologia como parâmetro. Este arquivo de topologia possui pelo menos um bloco INPUT e um OUTPUT. INPUT e OUTPUT informam ao SIMNT que o subsistema apresenta entradas e saídas externas. Na lista 3.11 temos o arquivo de topologia do sistema da figura 3.11. Neste caso LOOP e ADD são definidos como blocos STRUCT, e seus arquivos de parâmetros são LOOP.TOP (lista 3.12) e ADD.TOP (lista 3.13).

```
# LOOP.TOP

{ INPUT         INPUT
{ ADD1          ADD
{ FORK          FORK
{ DELAY         DELAY       DELAY.PAR
{ OUTPUT        OUTPUT

[ INPUT   0 ADD1   0 10000
[ ADD1    0 FORK   0 10000
[ FORK    0 DELAY  0 10000
[ DELAY   0 ADD1   1 10000
[ FORK    1 OUTPUT 0 10000
```

Lista 3.12 Arquivo de topologia LOOP.TOP para o sistema da figura 3.12.

```
#  ADD com três entradas

{ INPUT1        INPUT
{ INPUT2        INPUT
{ INPUT3        INPUT
{ ADD1          ADD
{ ADD2          ADD
{ OUTPUT        OUTPUT

[ INPUT1     0 ADD1      0 10000
[ INPUT2     0 ADD1      1 10000
[ INPUT3     0 ADD2      0 10000
[ ADD1       0 ADD2      1 10000
[ ADD2       0 OUTPUT    0 10000
```

Lista 3.13 Arquivo de topologia ADD.TOP para o sistema da figura 3.13.

Em LOOP.TOP temos uma entrada e uma saída externa, logo, um bloco INPUT e um OUTPUT. Já em ADD.TOP, que é um somador de três entradas, temos três INPUTs e um OUTPUT.

3.6 Executando Subsistemas

O construtor da classe *Block*, responsável pela interpretação dos arquivos de topologia, trata de maneira diferenciada os blocos STRUCT, INPUT e OUTPUT. Um bloco STRUCT tem um número variável de entradas e saídas, portanto, o construtor

carrega seu arquivo de topologia associado e determina estes parâmetros a partir do subsistema. Quando o construtor encontra um bloco INPUT, ele incrementa o número de entradas externas *ExtNin*, registra o número do bloco em *BlockExtNin* e o seu endereço de leitura em *AdressExtNin*. De maneira similar, o bloco OUTPUT possui *ExtNout*, *AdressExtNout*, *BlockExtNout* e o número máximo de amostras *ExtMax*.

SIMNT foi implementado como um objeto da classe *BlockOrder*. O arquivo fonte principal (função *main*) têm três linhas de comando que merecem destaque :

1. ***BlockOrder SIMNT(filetop);***
   Declara o objeto com parâmetro de inicialização filetop (arquivo de topologia) e chama todas as classes da cadeia de herança - Block, BlockLib, BlockParam, BlockOut, BlockUser e BlockOrder.
2. ***SIMNT.Exec();***
   Chama a função-membro Exec da classe BlockOrder e inicia a execução dos blocos.
3. ***SIMNT.PutDataOut();***
   Chama a função-membro PutDataOut da classe BlockOut e grava os arquivos de saída.

A execução de um bloco STRUCT corresponde a execução de um outro objeto *BlockOrder*, interno ao SIMNT, onde seu arquivo de topologia apresenta entradas e saídas externas, pois é formado por INPUTs e OUTPUTs. STRUCT possui um objeto BlockUser inicializado com o arquivo de topologia do subsistema, entretanto, antes da chamada à função *Exec* ele recebe as amostras e suas características que são colocadas nos blocos INPUT, a partir dos seus números (*BlockExtNin*) e endereços de leitura (*AdressExtNin*). Depois da chamada à função Exec, que executa os blocos do subsistema de maneira idêntica ao sistema principal, STRUCT escreve nas saídas do bloco as amostras geradas e suas características a partir dos números dos blocos OUTPUT e seus endereços de escrita externos.

Para o sistema da figura 3.13 a ordem inicial de execução dos blocos é a seguinte: *DATA1 - DATA2 - DATA3- LOOP - AMPL - ADD.* Quando o bloco LOOP é executado (figura 3.12), o objeto BlockUser interno é inicializado com o arquivo LOOP.TOP, e a ordem da primeira execução é a seguinte: INPUT - ADD1 - FORK - DELAY - OUTPUT. Antes da execução são fornecidas as 256 amostras para INPUT, a simulação do subsistema termina quando todos os blocos internos retornarem DEAD ou STOP.

O sistema da figura 3.13 foi otimizado em velocidade, pois o bloco LOOP recebe de DATA1 256 amostras de uma só vez, e gera outras 256 amostras para ADD, já no caso do sistema da figura 3.2, a realimentação introduzida por DELAY gerava apenas uma amostra de cada vez na saídas de FORK, sendo necessárias 256 chamadas aos blocos ligados a estas saídas (ADD2 e ADD3). Isto significa um ganho de desempenho em sistemas com muitos blocos e grande número de amostras.

Além disso, a figura 3.15 representa a implementação de um somador de três entradas, sem necessidade de programação.

Capítulo 4

Simulação de Sistemas Ópticos

4.1 Introdução

Neste capítulo, apresentamos algumas aplicações do SIMNT para desenvolvimento e análise de sistemas de comunicação óptica[1][2]. Estes sistemas são do tipo IM/DD (intensidade modulada/detecção direta). Os dois primeiros sistemas utilizam a arquitetura da figura 4.1, sendo formados por um driver de corrente, um laser monomodo de realimentação distribuída (DFB), um canal de fibra óptica, e um receptor, modelado como fotodetector PIN seguido de filtro de formato gaussiano. O terceiro sistema utiliza a arquitetura da figura 4.2[3], onde serão acoplados oito canais em uma fibra óptica, sendo realizada a sintonia de um destes canais utilizando um filtro de Fabry-Perot. Inicialmente apresentaremos a implementação e o modelamento matemático dos dispositivos utilizados para estas simulações. Como o objetivo deste capítulo é mostrar a utilização do SIMNT para simular sistemas em geral, de arquiteturas diversas e com grande número de blocos, os modelos adotados são matematicamente simples. Uma versão do SIMNT dedicada a simulação de sistemas comerciais exigirá maior rigor quanto aos modelamentos matemáticos dos dispositivos.

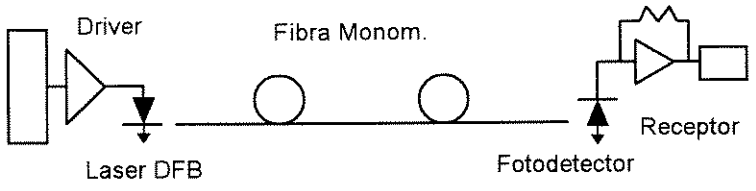


Figura 4.1 Arquitetura de um sistema óptico monocanal

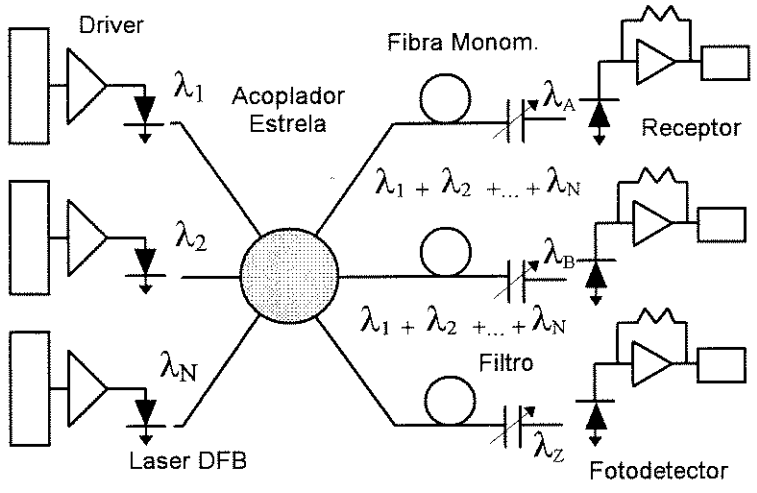


Figura 4.2 Arquitetura típica de um sistema de N canais WDM/WDMA.

4.2 Implementação e Modelamento dos Dispositivos

Nesta seção apresentamos os modelos utilizados para a simulação dos sistemas definidos nas figuras 4.1 e 4.2.

4.2.1 Driver de Corrente

O driver de corrente é registrado na biblioteca de modelos com o nome *DRIVERNRZ*, e é responsável pela geração dos pulsos de corrente a serem injetados em dispositivos como o laser. A corrente injetada no laser[4] é dada por(4.1)

$$I(t) = I_{bias} + \sum_{k=-\infty}^{\infty} A_k I_p(t - kT) \qquad (4.1)$$

onde I_{bias} é a corrente de polarização; $\{A_k\}$ apresenta uma sequência de bits independentes e identicamente distribuídos, para a qual $\Pr\{A_k = 1\} = \Pr\{A_k = 0\} = 0.5$, para todo k, onde $\Pr\{A\}$ denota a probabilidade do acontecimento de A; T é o período de bit $(taxa\ de\ bits)^{-1}$; e I_p é o pulso de corrente.

O formato do pulso de corrente[3] é dado pela seguinte equação,

$$I_p(t) = \begin{cases} 0, & t < 0 \\ I_m(1 - e^{-\frac{2.2t}{\tau_r}}), & 0 < t < T \\ I_m e^{-\frac{2.2(t-T)}{\tau_r}}, & t > T \end{cases} \qquad (4.2)$$

onde τ_r é o tempo de subida do pulso (entre 10 e 90 % de T) e I_m é a corrente de modulação (valor de pico).

Por esta definição, $I(t)$ tem o formato NRZ (não retorno a zero)[1].

O arquivo de parâmetros fornecido ao DRIVERNRZ, para geração dos dados da figura 4.3, é apresentado na lista 4.1.

```
# Arquivo de parâmetros para DRIVERNRZ

double   IBias 38.86e-3          # corrente de polarização (A)
double   IMod 23.115e-3          # valor de pico da corrente de modulação(A)
double   BitRate 3e9             # bitrate (bps)
double   RiseTime 166.666e-12    # tempo de subida/descida (s)
int   Length  10                 # comprimento da sequência de bits
int   Seq   0 0 1 0 1 0 0 0 0 0  # sequência de bits
```

Lista 4.1 Arquivo de parâmetros fornecido a um bloco DRIVERNRZ.

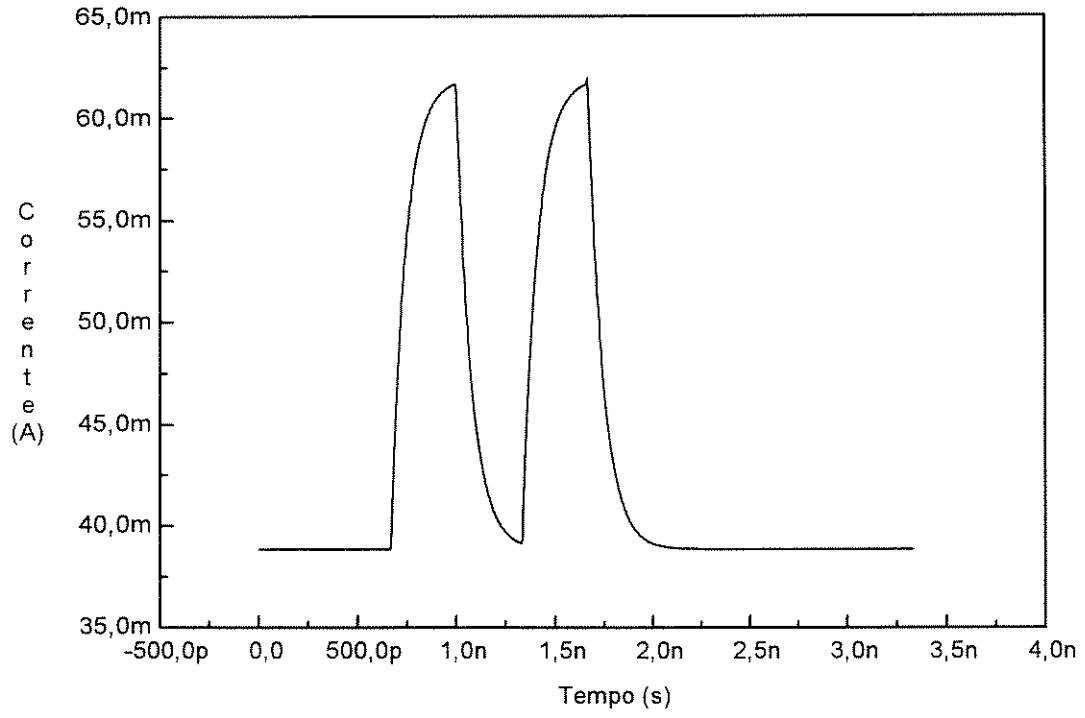


Figura 4.3 Corrente na saída do bloco DRIVERNRZ para os parâmetros da lista 4.1.

4.2.2 Laser DFB Monomodo

O termo *diodo laser monomodo* é aplicado a um tipo modificado de estrutura do laser de Fabry-Perot de duas faces. O laser monomodo mais utilizado atualmente é o laser de realimentação distribuída (DFB)[2], neste dispositivo é colocada uma grade na região ativa que atua como um filtro ressonante, suprimindo os modos laterais de emissão.

O laser DFB é registrado na biblioteca de modelos como *SMLASER* e possui uma entrada e uma saída. Em sua entrada o *SMLASER* recebe uma sequência de amostras de corrente, e em sua saída gera o campo elétrico $E(t)$.

A potência óptica do laser semicondutor em função da corrente injetada $I(t)$, é determinada pela equações de taxa[5], que descrevem a relação entre a densidade de fótons $p(t)$, densidade de portadores $n(t)$ e a fase óptica $\phi(t)$ dentro da cavidade do laser. Estas equações são dadas por

$$\frac{dp(t)}{dt} = \Gamma G(t)(n(t) - n_0)p(t) - \frac{p(t)}{\tau_p} + \frac{\beta \Gamma n(t)}{\tau_n} \tag{4.3}$$

$$\frac{dn(t)}{dt} = \frac{I_p(t)}{qV_a} - G(t)(n(t) - n_0)p(t) - \frac{n(t)}{\tau_n} \tag{4.4}$$

$$\frac{d\phi(t)}{dt} = \frac{\alpha}{2}\left(\Gamma v_g a_0 (n(t) - n_0) - \frac{1}{\tau_p}\right) \tag{4.5}$$

Nestas equações, Γ é o fator de confinamento, n_0 é a densidade de portadores na transparência, τ_p é o tempo de vida médio do fóton, β é a fração de emissão espontânea acoplada ao modo de emissão, τ_n é o tempo de vida médio do elétron, q é a carga do elétron, V_a é o volume da região ativa, v_g é a velocidade de grupo, a_0 é o coeficiente de ganho e α é o fator de largura de linha. O parâmetro $G(t)$ é o coeficiente de ganho material, o qual é definido por

$$G(t) = \frac{v_g a_0}{1 + \varepsilon p(t)} \tag{4.6}$$

onde ε é o fator de supressão do ganho. Por integração numérica de (4.3)-(4.5) usando a corrente injetada por (4.1), obtêm-se a potência óptica $P(t)$, emitida por face, a qual é, então, dada por

$$P(t) = \frac{p(t)}{2}\frac{V_a \eta_0 h f_0}{\Gamma \tau_p} \tag{4.7}$$

onde η_0 é a eficiência quântica diferencial do laser, h é a constante de Planck, e f_0 é a frequência de emissão. O campo elétrico na saída do laser é, assim, dado por $E(t)$ [6]:

$$E(t) = \sqrt{P(t)}e^{j(2\pi f_0 t + \phi(t))} \tag{4.8}$$

O algoritmo de Runge-Kutta[7], de quarta ordem com passo adaptivo é empregado para a integração numérica das equações de taxa. O passo adaptivo necessita de um maior tempo de processamento, mas de um menor número de amostras. Utilizando os valores de corrente da figura 4.3 e os parâmetros fornecidos ao SMLASER, e apresentados na lista 4.2, obtêm-se os resultados apresentados na figura 4.4, que representam a potência óptica na saída do laser. A figura 4.5 representa a potência óptica na saída do laser, para uma entrada dada pela sequência 0111100000, gerada no driver de corrente. Ambos os dispositivos foram definidos com os mesmos parâmetros anteriores. Estes resultados estão de acordo com os apresentados nas figuras 5 e 6 da referência[3].

```
# Arquivo de parâmetros para o Laser

double CenterFreq 1.935483871e14 # frequencia central de emissão (Hz)
double Gama       0.4            # fator de confinamento do modo de emissão
double n0         1e18           # densidade de eletrons na transparência (cm^-3)
double tp         3e-12   # tempo de vida médio do fóton (s)
double Beta       3e-5    # fração de emissão espontânea acop. ao modo da emiss.
double tn         1e-9    # tempo de vida médio do elétron (s)
double Va         1.5e-10 # volume da camada ative (cm^-3)
double Alpha      5       # fator de largura de linha
double Vg         8.5e9   # velocidade de grupo na região ativa (cm/s)
double a0         2.5e-16 # coeficiente de ganho diferencial (cm^2)
double Eta0       0.4     # eficiência quântica diferencial total
double e          1e-17   # fator de supressão do ganho
```

Lista 4.2 Arquivo de parâmetros para o bloco SMLASER.

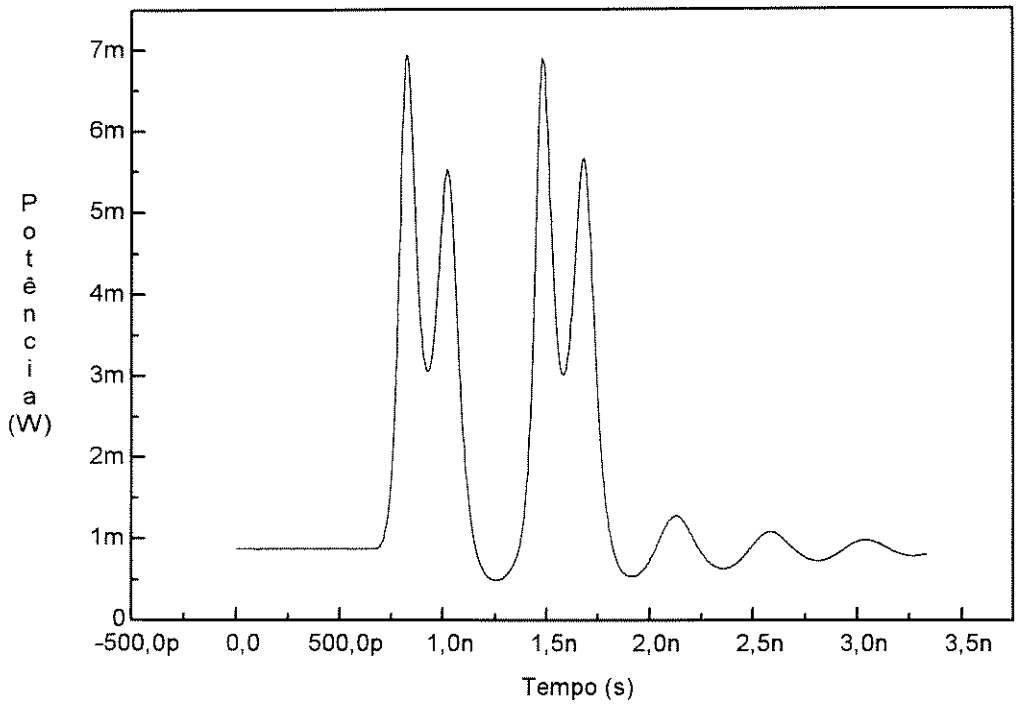


Figura 4.4 Potência óptica na saída do laser para a sequência 0010100000.

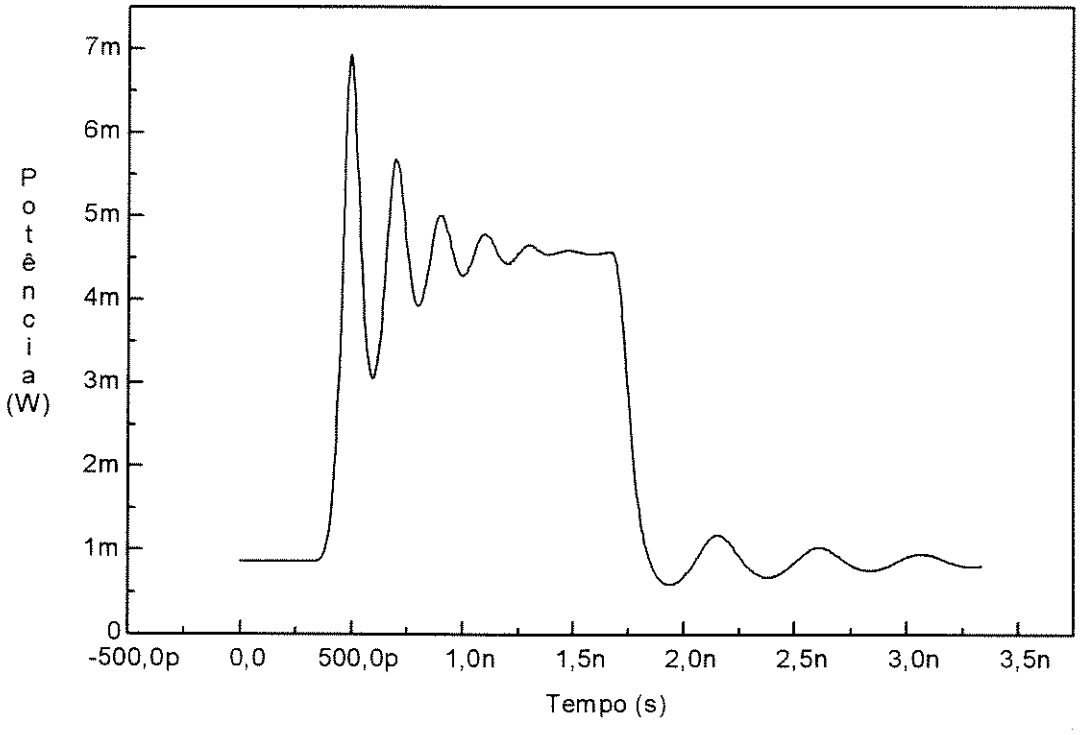


Figura 4.5 Potência óptica na saída do laser para a sequência 0111100000.

4.2.3 Fibra Monomodo

A fibra utilizada nos exemplos aqui tratados foi a fibra monomodo. A fibra monomodo permite maiores distâncias de transmissão, não apresentando o efeito da dispersão modal[1] (presente nas fibras multimodo).

Ela foi registrada na biblioteca de modelos como *FIBER1*, e possui uma entrada e uma saída. O campo elétrico na saída da fibra no domínio da frequência, $S_{fibra}(f)$, é dado por[3]

$$S_{fibra}(f) = H_{fibra}(f)S_s(f) \tag{4.9}$$

onde

$$H_{fibra}(f) = e^{j\frac{\pi\lambda^2 Dlf^2}{c}} \tag{4.10}$$

é a função de transferência de uma fibra monomodo; $S_s(f)$ é a transformada de fourier do campo elétrico na entrada da fibra; l é o comprimento; λ é o comprimento de onda; c é a velocidade da luz no vácuo; e D é a dispersão na fibra. Na saída temos o campo elétrico como a transformada inversa de $S_{fibra}(f)$.

A lista 4.3 fornece os parâmetros típicos para *FIBER1*.

```
# Arquivo de parâmetros para FIBER1
double Len    50        # distância (Km)
double Aten   0.25      # atenuação (dB/Km)
double Disp   17        # dispersao ps/(nm.Km)
double Lambda 1.53e-6 # comprimento de onda (m)
```

Lista 4.3 Parâmetros típicos para o bloco *FIBER1*.

4.2.4 Fotodetector PIN

O fotodetector converte a luz incidente em corrente. Pela facilidade de modelamento matemático do dispositivo, utilizamos nos nossos exemplos o fotodiodo PIN[2].

A fotocorrente, gerada por um fotodetector PIN, é relacionada com a potência óptica incidente $P(t)$ por

$$i(t) = \frac{\eta}{hf} P(t) + \lambda_0 \tag{4.11}$$

onde η é a eficiência quântica do fotodetector e λ_0 a corrente de escuro. $P(t)$ pode ser calculada a partir do campo elétrico incidente S_i por

$$P(t) = S_i(t) S^*_i(t) \tag{4.12}$$

O nome do bloco na biblioteca é PIN1, e seu arquivo de parâmetros é apresentado na lista 4.4.

```
# Arquivo de parâmetros para PIN1

double Eta    0.8       # eficiência quântica do fotodetector
double L0     10e-9     # corrente de escuro (A)
```

Lista 4.4 Parâmetros para o bloco *PIN1*.

4.2.5 Acoplador

Como o SIMNT associa a cada buffer de dados um vetor de amostras, e suas características: a frequência de amostragem, o tempo inicial e a frequência central do envelope complexo, a soma de dois sinais de frequências centrais diferentes exige um tratamento específico. Deste modo, o acoplador, registrado como *ACOPFREQ* possui

duas entradas e uma saída, onde as frequências central e de amostragem de saída são calculadas a partir dos valores das frequências de entrada. Desta forma teremos

$$f_{out} = \frac{f_0 + f_1}{2} \tag{4.13}$$

$$f_s = Max(f_0 + \frac{f_{s0}}{2}, f_1 + \frac{f_{s1}}{2}) - Min(f_0 - \frac{f_{s0}}{2}, f_1 - \frac{f_{s1}}{2}) \tag{4.14}$$

onde

f_{out} = frequência central de saída,

f_0, f_1 = frequências centrais de entrada,

f_s = frequência de amostragem de saída,

f_{s0}, f_{s1} = frequências de amostragem de entrada.

4.2.6 Filtro de Formato Gaussiano

Um filtro tem como objetivo aumentar a relação sinal/ruído, preservando as características essenciais do sinal. O filtro utilizado aqui tem formato gaussiano e a sua função de transferência é dada por

$$H(f) = e^{-\pi(\frac{f}{2f_R})^2} \tag{4.15}$$

onde $f_R = b / T$, T é o período de bit e b um valor tipicamente variando entre 0.6 e 0.8. O bloco é registrado na biblioteca de modelos como GAUSSFILT e o arquivo de parâmetros é fornecido na lista 4.5.

```
# Arquivo de parâmetros

double BitRate 3e9  # bitrate (bps)
double R  0.75  # fr=2*R*BitRate
```

Lista 4.5 Arquivo de parâmetros para GAUSSFILT.

4.2.7 Filtro de Fabry-Perot

Em sistemas de arquitetura WDM (figura 4.2), a seleção de um ou mais canais é realizada através de filtros sintonizáveis. O dispositivo utilizado nos nossos exemplos é o filtro de Fabry-Perot[8]. O campo elétrico na saída do filtro de Fabry-Perot no domínio da frequência é dado por

$$S_{FP}(f) = H_{FP}(f)S_s(f) \tag{4.16}$$

onde H_{FP} é a função de transferência do filtro, dada por

$$H_{FP}(f) = \frac{T_p}{1 - R_p e^{j2\pi\frac{(f-fc)}{FSR}}} \tag{4.17}$$

nesta última equação T_p e R_p são os coeficientes de transmissão e reflexão de potência do filtro, respectivamente; o parâmetro f_c é a frequência central do filtro; e FSR é *o free spectral range* definido em termos da banda de 3-dB $FWHM$ (*full width at half maximum*) como:

$$FWHM = \frac{FSR}{F} \tag{4.18}$$

onde F é a finesse do filtro, sendo definida por

$$F = \frac{\pi\sqrt{R_p}}{1 - R_p} \tag{4.19}$$

O nome do bloco na biblioteca de modelos é *FPFILTER* e seu arquivo de parâmetros está na lista 4.6.

```
# parâmetros para o filtro de Fabry-Perot

double FWHM       20e9              # banda de meia potência do filtro (Hz)
double FSR        3.7e12            # Free Spectral Range (Hz)
double ChannelFreq  1.935483871e14  # frequência do canal sintonizado (Hz)
```

Lista 4.6 Arquivo de parâmetros para *FPFILTER*.

4.3 Simulação de Enlace Óptico I

Nesta seção, utilizamos o SIMNT para analisar a transmissão através do enlace óptico apresentado na figura 4.6. As próximas figuras (4.7-4.11) apresentam a evolução do campo elétrico nos diversos pontos do enlace óptico. Os parâmetros utilizados para a simulação foram

$$I_{bias} = 38.86mA$$
$$I_{\mathrm{mod}} = 23.11mA$$
$$B = 1/T = 3.33Gb/s$$
$$\tau_r = 0.5T$$
$$FWHM = 20GHz$$
$$FSR = 3.7THz$$
$$\lambda_c = 1.55\mu m$$

onde λ_c é o comprimento de onda de emissão.

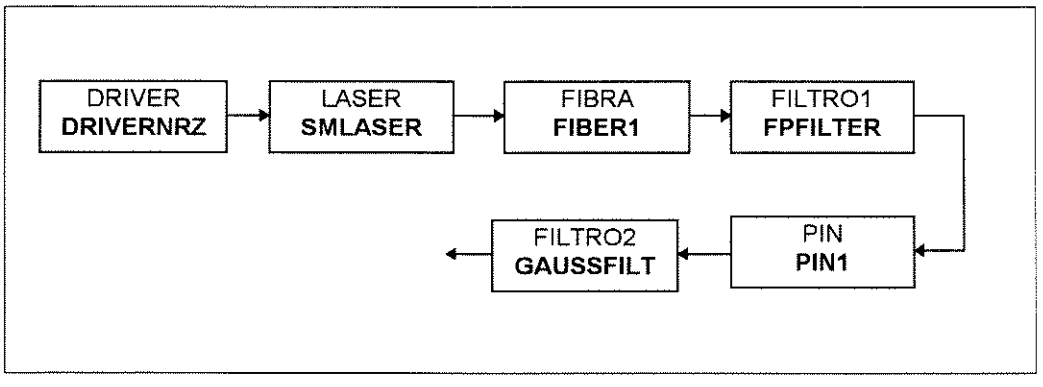


Figura 4.6 Diagrama de blocos de um enlace óptico.

Na figura 4.7 temos a corrente na saída do driver. O formato acentuado dos picos de corrente é devido ao valor de τ_r. Podemos observar que o valor máximo da corrente é a soma do pico da corrente de modulação com a corrente de polarização ($I_{max} = I_{bias} + I_{\mathrm{mod}} \approx 62mA$). Nesta figura, utilizamos uma sequência de 32 bits, numa taxa de 3.33 *Gb/s*, onde os primeiros 8 bits desta sequência são 01010101.

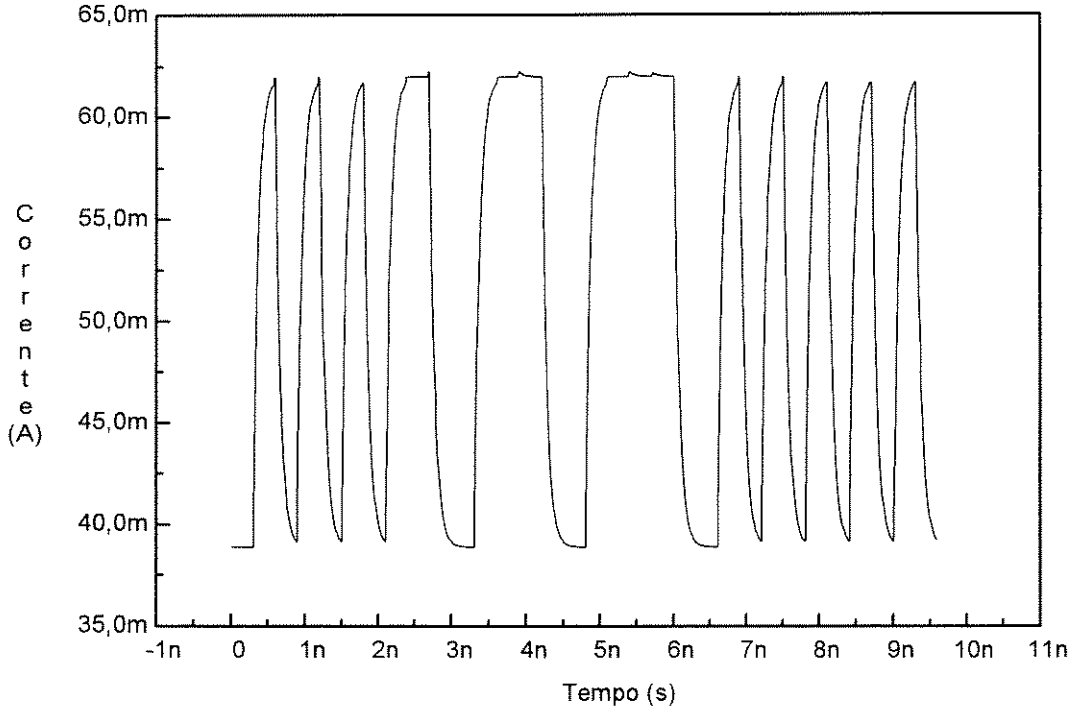


Figura 4.7 Corrente na saída do bloco DRIVER.

A figura 4.8 mostra a potência óptica na saída do laser, utilizando a corrente injetada pelo driver (figura 4.7), onde observamos a dependência das oscilações de relaxação[5] com o formato da sequência de bits. No intervalo entre 2 e 3 nanosegundos (0110) a amplitude das oscilações é superior do que no intervalo entre 4.5 e 6.5 nanosegundos (011110). Quando a sequência de bits apresenta um comportamento estacionário, observamos que os oscilações ficam estáveis (intervalo entre 7.5 e 9.5 nanosegundos (010101)).

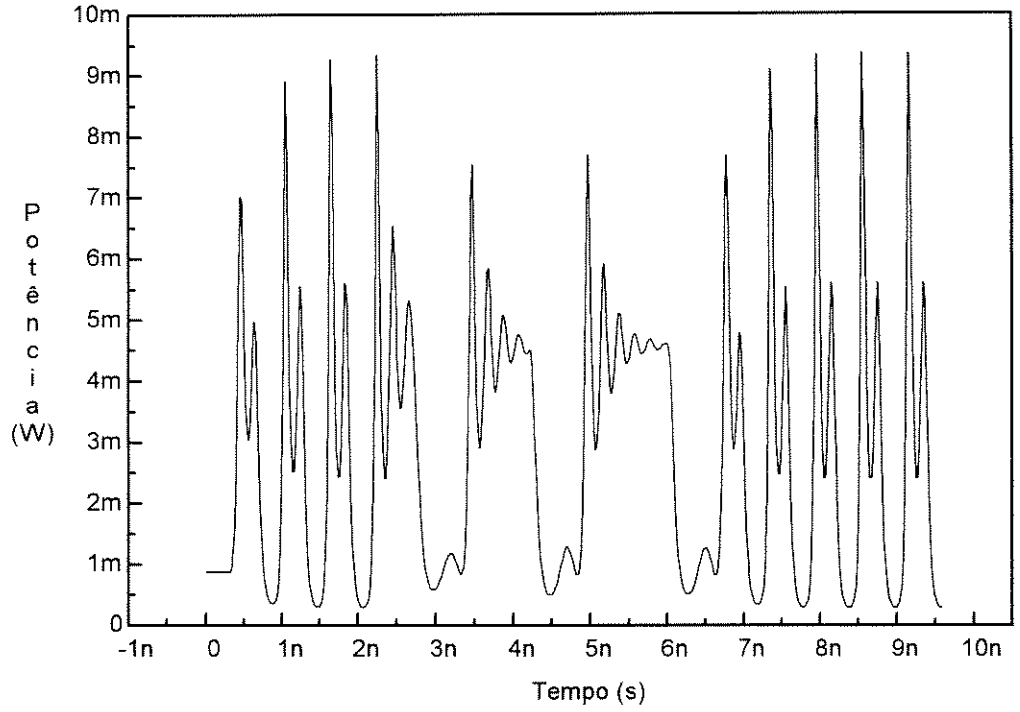


Figura 4.8 Potência óptica na saída do bloco LASER.

A figura 4.9 apresenta a potência na saída da fibra óptica (50 quilômetros de comprimento). O alargamento dos pulsos é devido ao efeito de dispersão cromática[1] da fibra, um fator limitante na transmissão a longas distâncias. Cabe aqui uma observação, o efeito do *chirp*[5] desloca o espectro de frequência do laser, de modo que a frequência central de emissão pode não coincidir com a frequência onde a dispersão da fibra é mínima, contribuindo ainda mais para o alargamento dos pulsos.

Na figura 4.10 temos a potência óptica na saída do filtro de Fabry-Perot. A diferença de amplitude em relação a potência de entrada (figura 4.9), deve-se a pequena diferença entre o comprimento de onda do canal sintonizado (1.55 μm) e a frequência central do laser, deslocada devido ao *chirp*. Para uma banda menor do que 20 *GHz* podemos ter uma alteração ainda maior do formato dos pulsos.

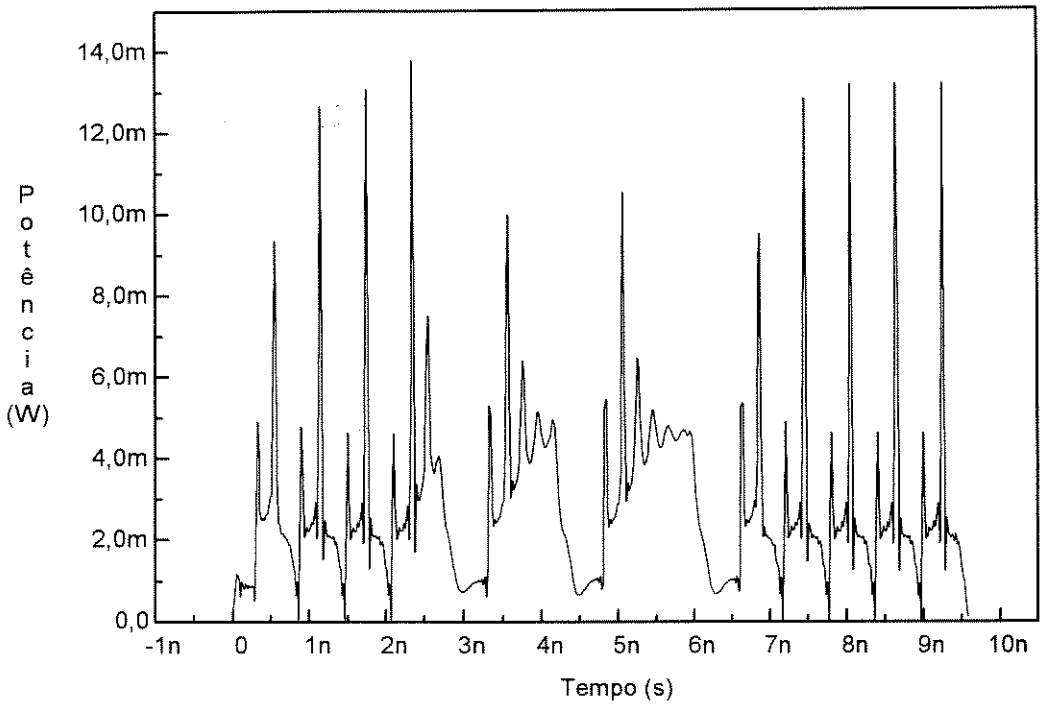


Figura 4.9 Potência óptica na saída do bloco FIBRA.

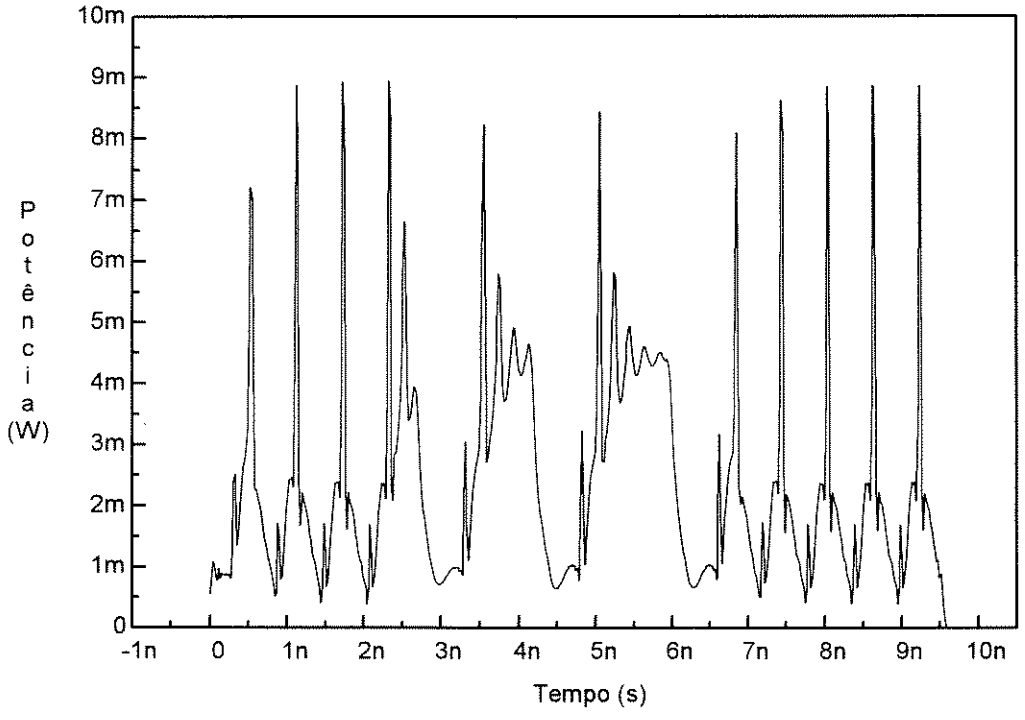


Figura 4.10 Potência óptica na saída do bloco FILTRO1.

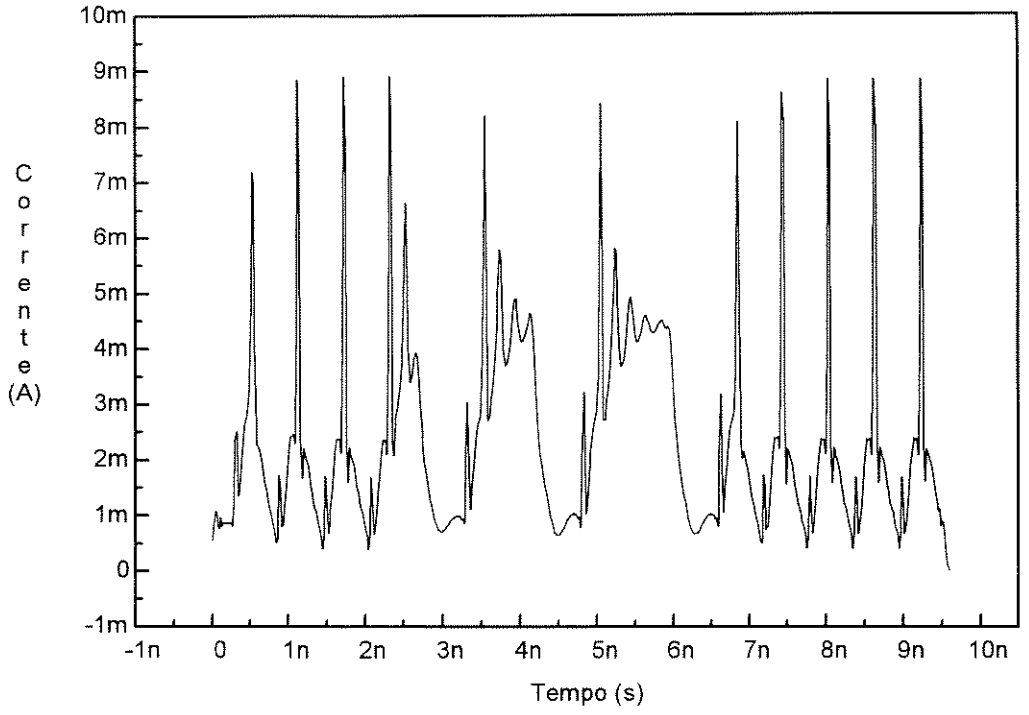


Figura 4.11 Corrente na saída do bloco PIN.

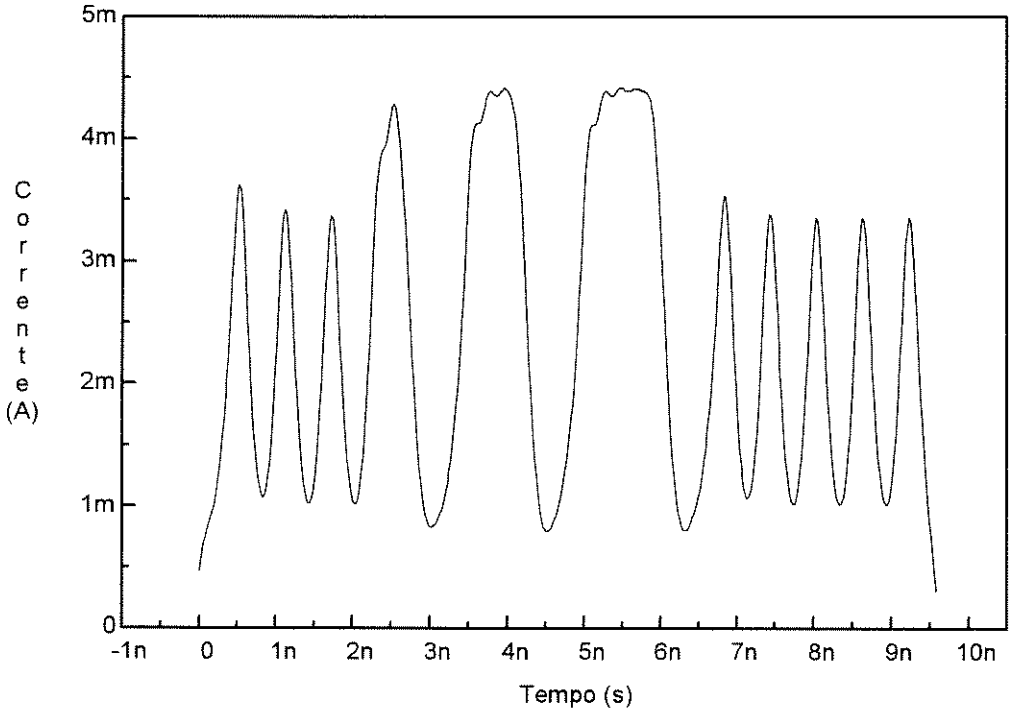


Figura 4.12 Corrente na saída do bloco FILTRO2.

A figura 4.11 representa a corrente na saída do fotodetector PIN. Estas amostras são então colocadas na entrada do filtro de formato gaussiano. A figura 4.12 apresenta a

saída do filtro gaussiano, onde observamos que as componentes de alta frequência foram eliminadas. Fazendo uma comparação com a figura 4.7, percebemos que a diferença de amplitude entre os picos de corrente, e o alargamento temporal destes, contribuem para o aumento da probabilidade de erro[9] numa etapa posterior de decisão.

4.4 Simulação de Enlace Óptico II

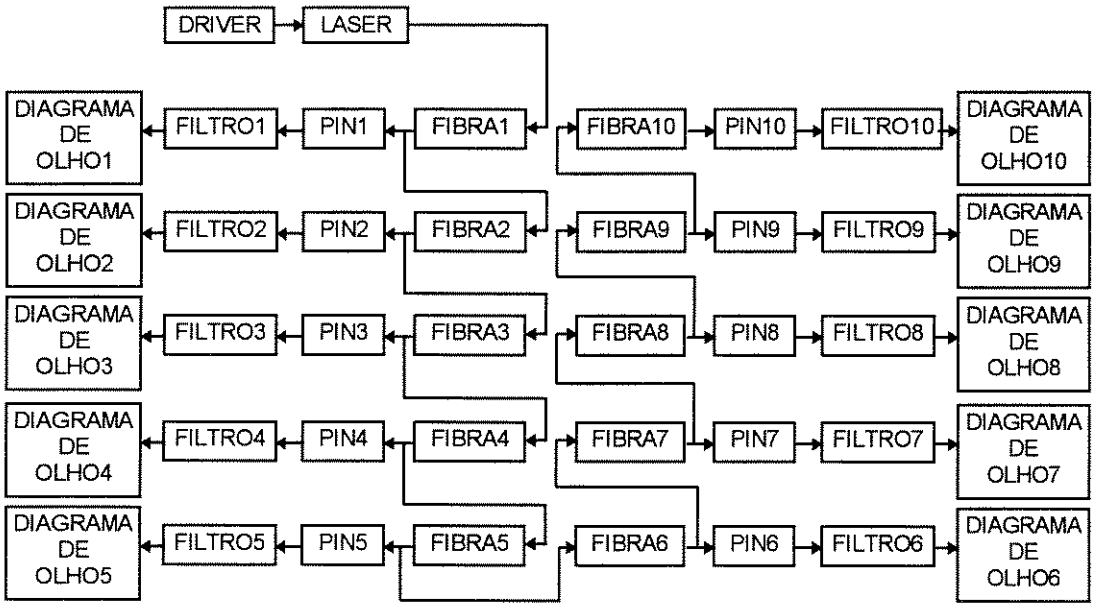


Figura 4.13 Diagrama de blocos de um enlace de 275 km de fibra óptica.

Vamos agora simular o enlace óptico definido pelo diagrama de blocos da figura 4.13. Neste diagrama, vamos avaliar o efeito da dispersão a cada 25 quilômetros de fibra óptica, com uma distância inicial de 50 quilômetros. Deste modo, são necessários 42 blocos, pois a cada trecho da fibra foram colocados um fotodetector PIN, um filtro de formato gaussiano, e um bloco para geração do diagrama de olho. Devemos destacar que os efeitos de atenuação foram desprezados. Os parâmetros fornecidos aos dispositivos são os seguintes:

$I_{bias} = 38.86mA$
$I_{\mathrm{mod}} = 23.11mA$
$B = 1/T = 3.33Gb/s$
$\tau_r = 0.5T$
$\lambda_c = 1.55\mu m$

Os gráficos a seguir (figuras 4.14-4.19) são os resultados do processo de simulação. Nestes, a influência da dispersão na abertura dos diagramas de olho[1] pode ser observada.

A figura 4.14 mostra o diagrama de olho para uma distância de 50 quilômetros, supondo uma região de decisão em torno de 250 picosegundos. Se o compararmos com a figura 4.15 (75 km), observaremos um pequeno fechamento do diagrama, entretanto, quando a comparação é em relação aos diagramas das figuras 4.16-4.18 observaremos que o diagrama vai se fechando até fechar completamente na figura 4.18.

A figura 4.19 é uma sobreposição dos valores na saída dos filtros, deste modo, o efeito da interferência intersimbólica[1] pode ser acompanhado em um intervalo de tempo (4.5-10 ns), desde a distância inicial de 50 quilômetros até os 275 quilômetros finais.

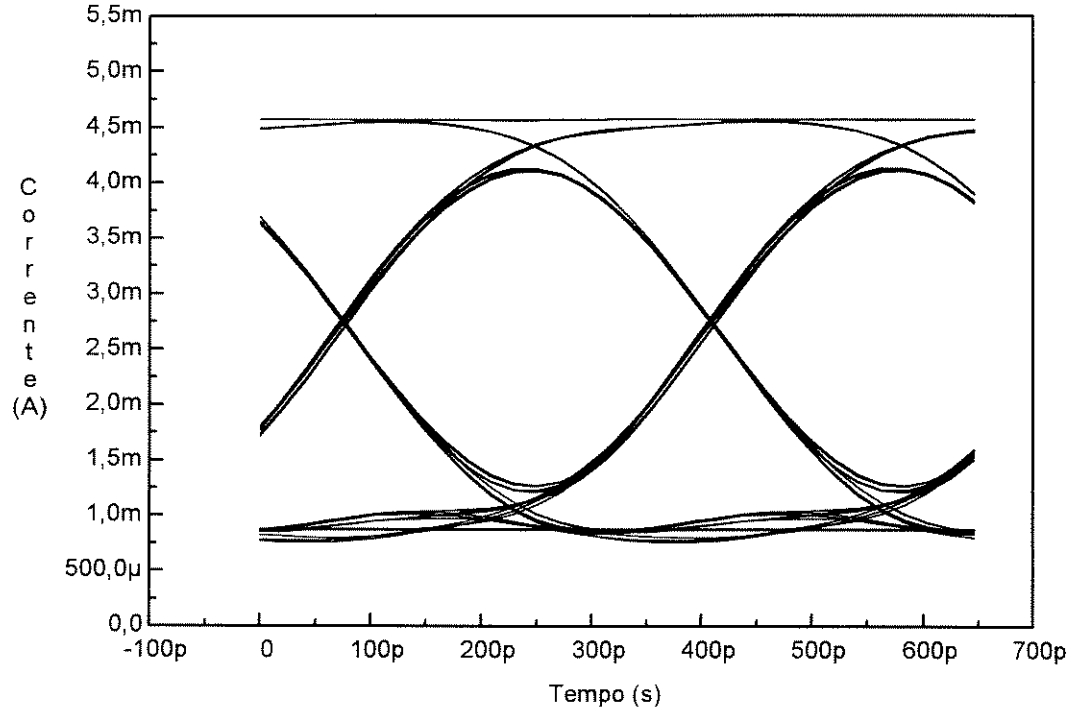


Figura 4.14 Diagrama de olho 1 - 50 km de fibra óptica.

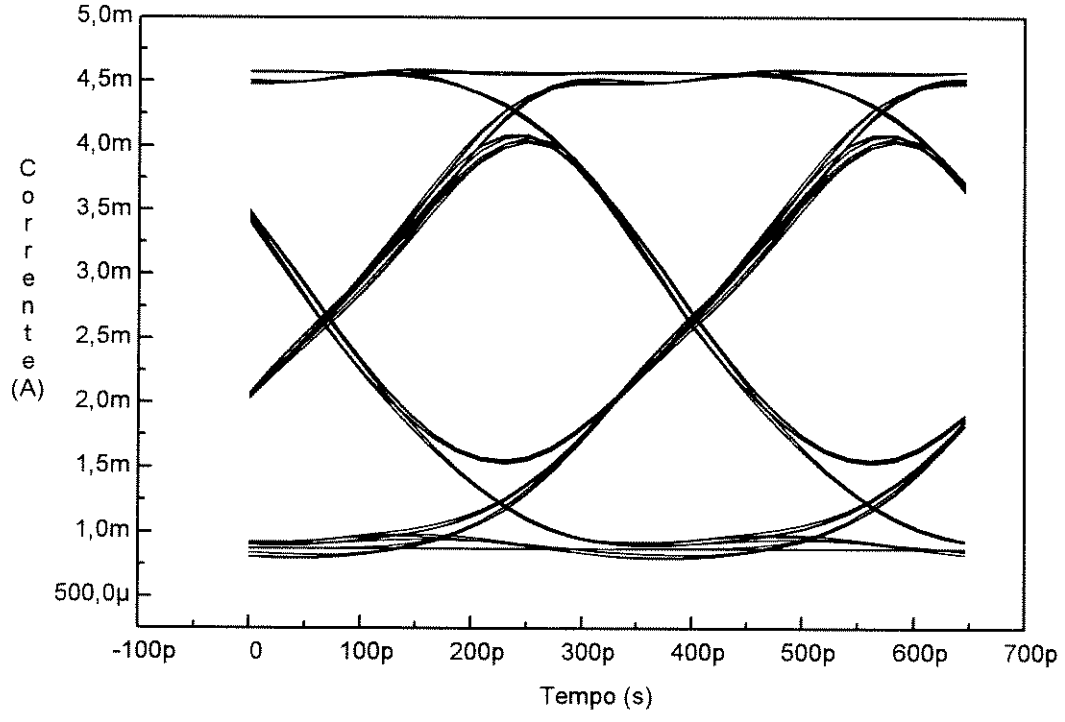


Figura 4.15 Diagrama de olho 2 - 75 km de fibra óptica.

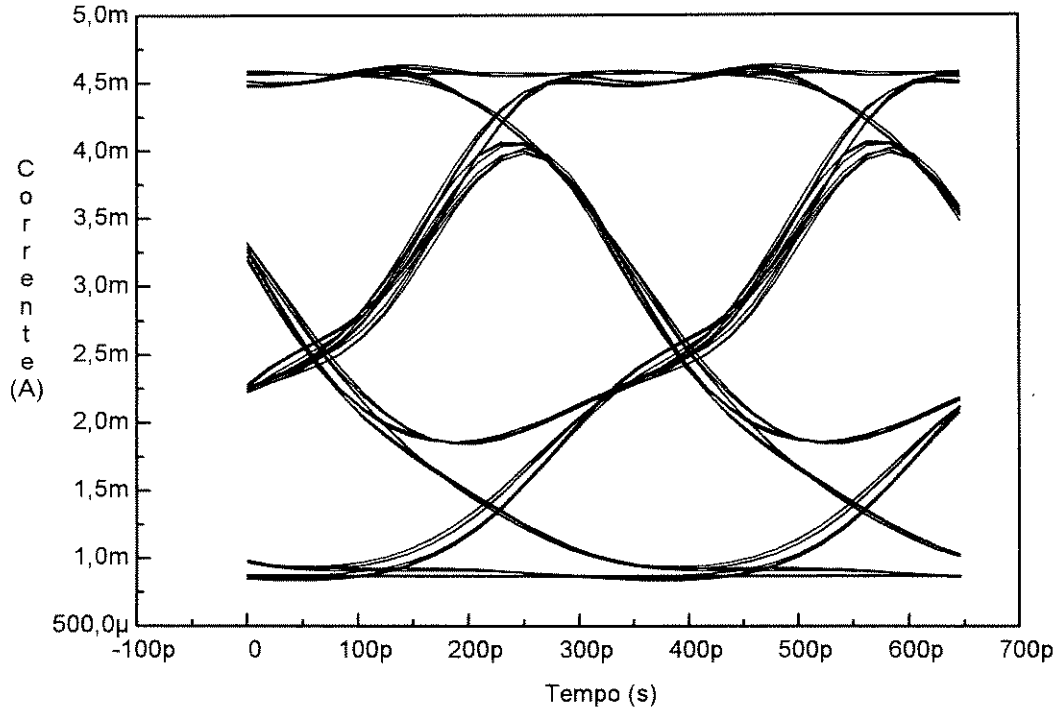


Figura 4.16 Diagrama de olho 3 - 100 km de fibra óptica.

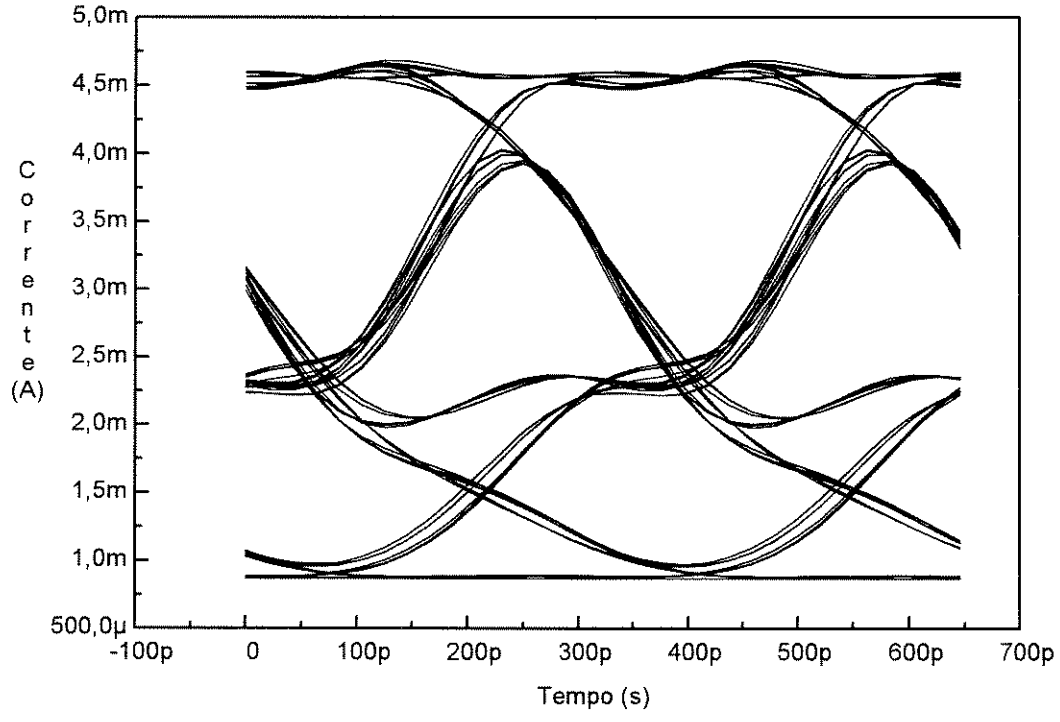


Figura 4.17 Diagrama de olho 4 - 125 km de fibra óptica.

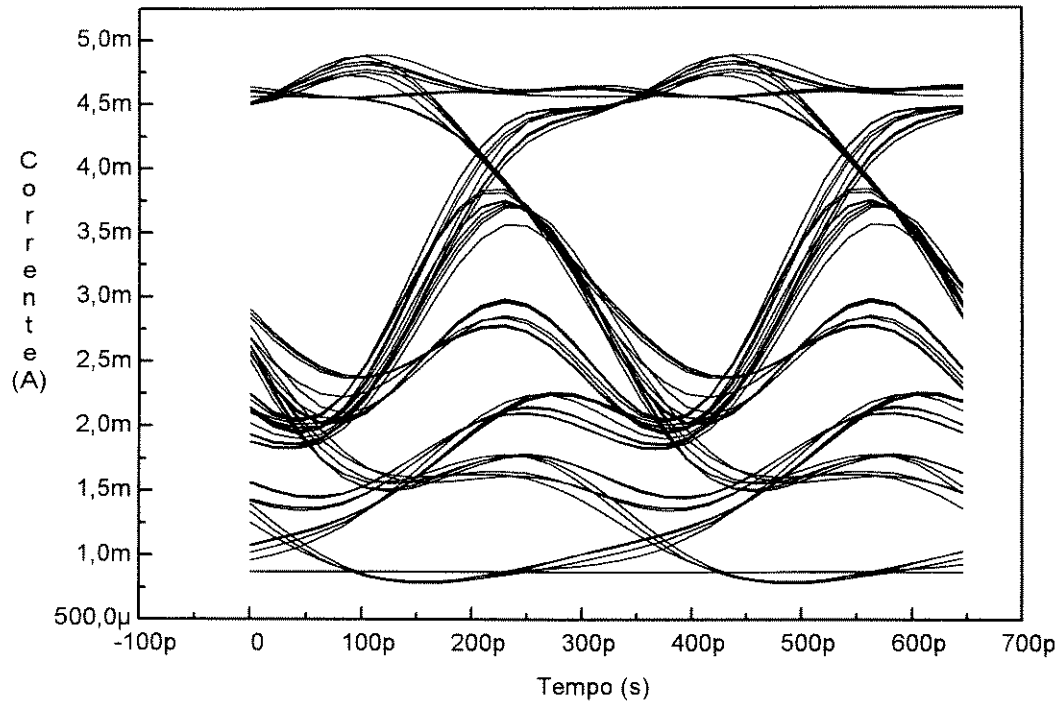


Figura 4.18 Diagrama de olho 5 - 150 km de fibra óptica.

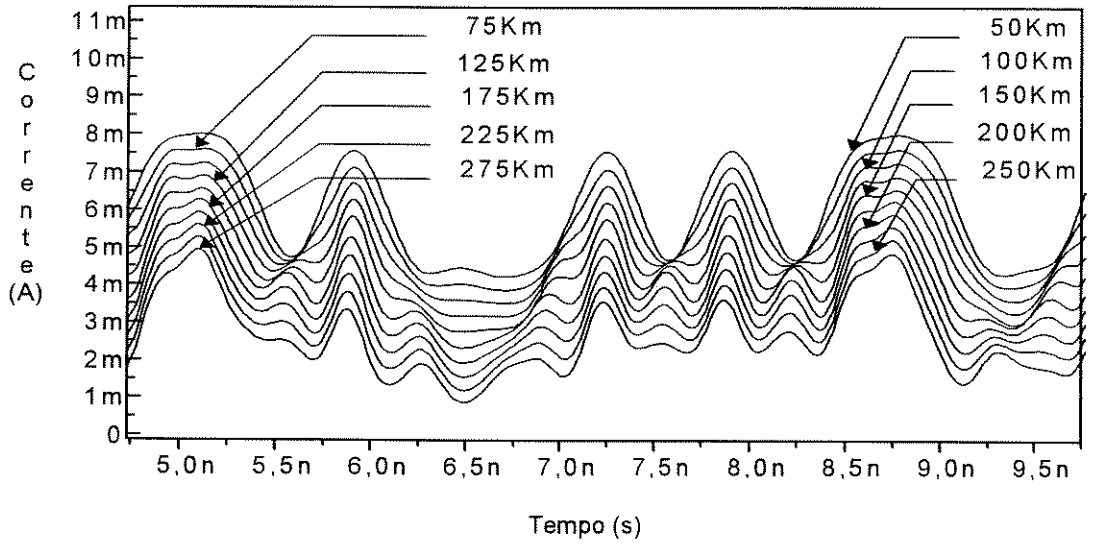


Figura 4.19 Janela de tempo ilustrando o efeito da dispersão.

4.5 Simulação de um Sistema WDM

O sistema a seguir utiliza dois drivers de corrente gerando sequências de bits complementares, que injetam corrente em oito lasers sintonizados em canais adjacentes, deste modo, podemos observar a interferência entre os canais em uma situação de pior caso[3]. Neste sistema, apresentaremos diagramas de olho para diversos valores de distância entre os canais e banda do filtro de Fabry-Perot, com os seguintes parâmetros fixos[2]:

$$I_{bias} = 37.52mA$$
$$I_{\mathrm{mod}} = 24.45mA$$
$$B = 1/T = 2Gb/s$$
$$\tau_r = 0.35T$$
$$FSR = 3.7THz$$
$$Canal = \frac{c}{1.55\mu} Hz$$

O valor adotado para FSR é igual à banda do amplificador óptico (~30nm), entretanto, este dispositivo não foi implementado, além disso, novamente foram desprezados os efeitos de atenuação em todo o sistema. Os valores de distância entre canais e *FHWM* são fornecidos na tabela 4.1.

Distância entre canais(GHz)	FWHM (GHz)				
50	30	25	20	15	10
25	25	20	15	10	
12	20	15	10		

Tabela 4.1 Valores utilizados para simulação do sistema 2.

Na figura 4.20 apresentamos o diagrama de blocos do sistema simulado. O bloco *Acoplador* é um subsistema, ou seja, formado por outros blocos (figura 4.20), deste modo obtivemos um acoplador de oito entradas. Na saída teremos somados todos os sinais provenientes dos oito lasers.

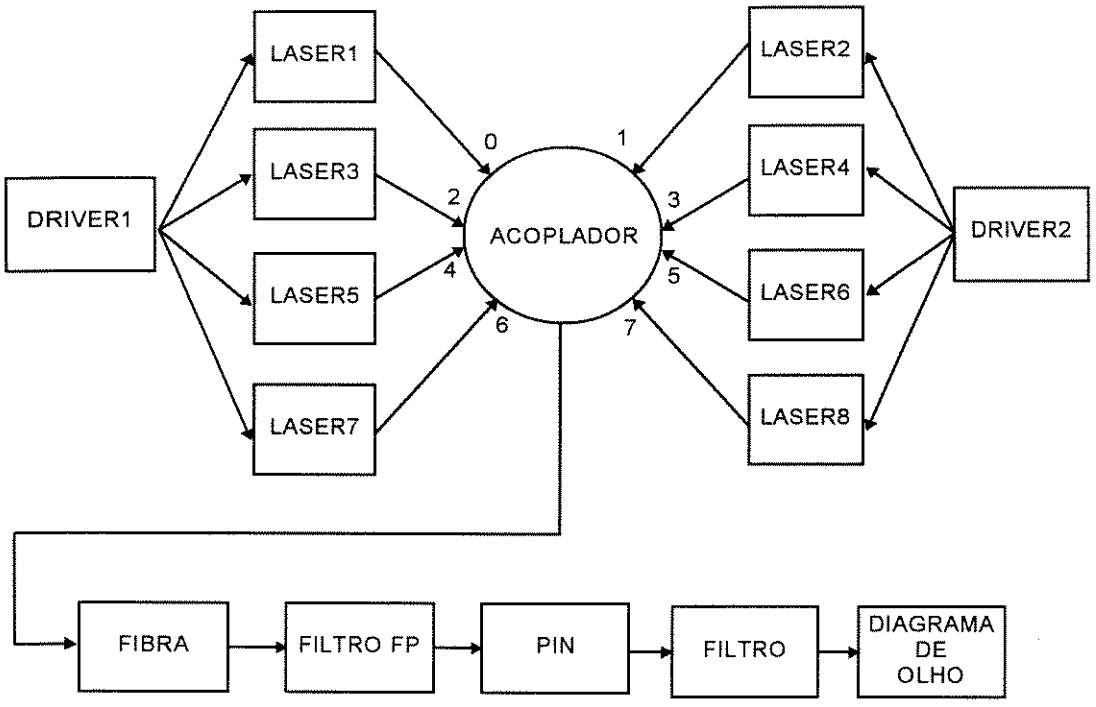


Figura 4.20 Diagrama de blocos de um sistema WDM 8 canais.

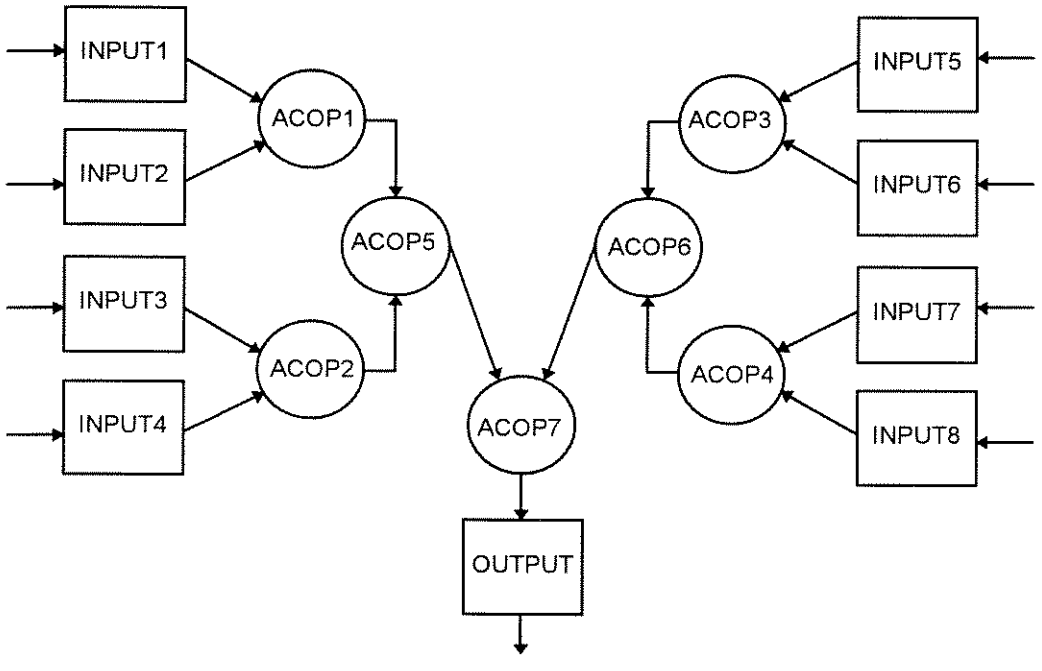


Figura 4.21 Diagrama de blocos de um acoplador de oito entradas

As figuras 4.22 e 4.23 mostram a corrente na saída dos blocos DRIVER1 e DRIVER2 respectivamente, associadas a sequências complementares de bits. Esta corrente é injetada nos blocos LASER (figura 4.20), de modo que os canais adjacentes tenham sequências diferentes. As figuras 4.24 e 4.25 apresentam o espectro do campo elétrico na saída dos lasers, onde podemos observar que os harmônicos de maior amplitude se encontram deslocados da frequência central *fc* de emissão em ambos os casos. Este deslocamento prejudica o desempenho do sistema, pois favorece a dispersão cromática na fibra óptica, além de necessitar uma maior banda para os filtros, de modo a acomodar o sinal entre suas frequências de corte.

As figuras 4.26-4.28 mostram os espectro do campo elétrico para diferentes espaçamentos entre os canais de transmissão dos lasers (12, 25 e 50 GHz). Nestas figuras podemos identificar os espectros das figuras 4.24 e 4.25, indicando a correta operação do acoplador de oito canais. Aparentemente, a figura 4.26 ilustra o caso mais crítico, onde a interferência entre canais adjacentes é visível, já nas figuras 4.27 e 4.28 é nítida a distância que separa os canais, 25 e 50 GHz respectivamente. Se utilizarmos toda a banda de frequência do amplificador óptico (3.7 THz) estaríamos transmitindo 74 canais simultaneamente, utilizando o espaçamento de 50 GHz, 148 no segundo caso e finalmente, 308 canais utilizando 12GHz de espaçamento.

As figuras 4.29-4.38 mostram diversos espectros do campo elétrico na saída do filtro de Fabry-Perot, sintonizado no comprimento de onda de 1.55μm, correspondente ao quarto canal (LASER4, também caracterizado pelas figuras 4.23 e 4.25). Como era esperado, o aumento gradual da banda do filtro de Fabry-Perot (FWHM), permite um aumento da contribuição dos canais adjacentes no canal selecionado, entretanto, são nas figuras 4.29-4.31 (espaçamento de 12GHz) onde este efeito é mais pronunciado.

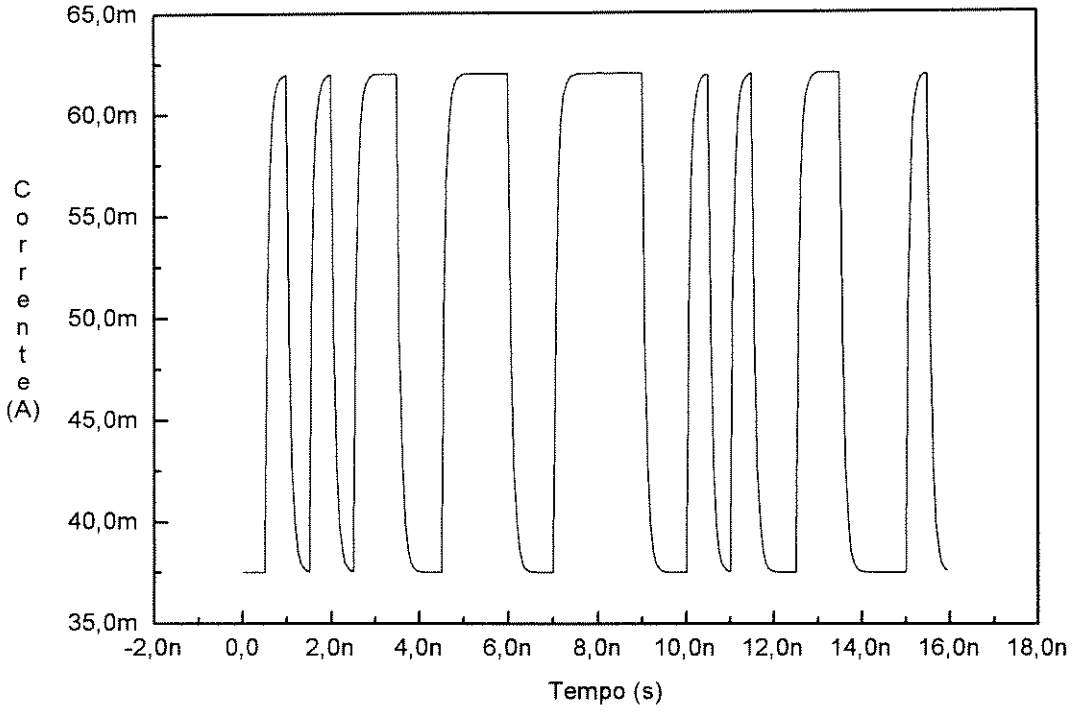


Figura 4.22 Corrente injetada em LASER1, 3, 5 e 7.

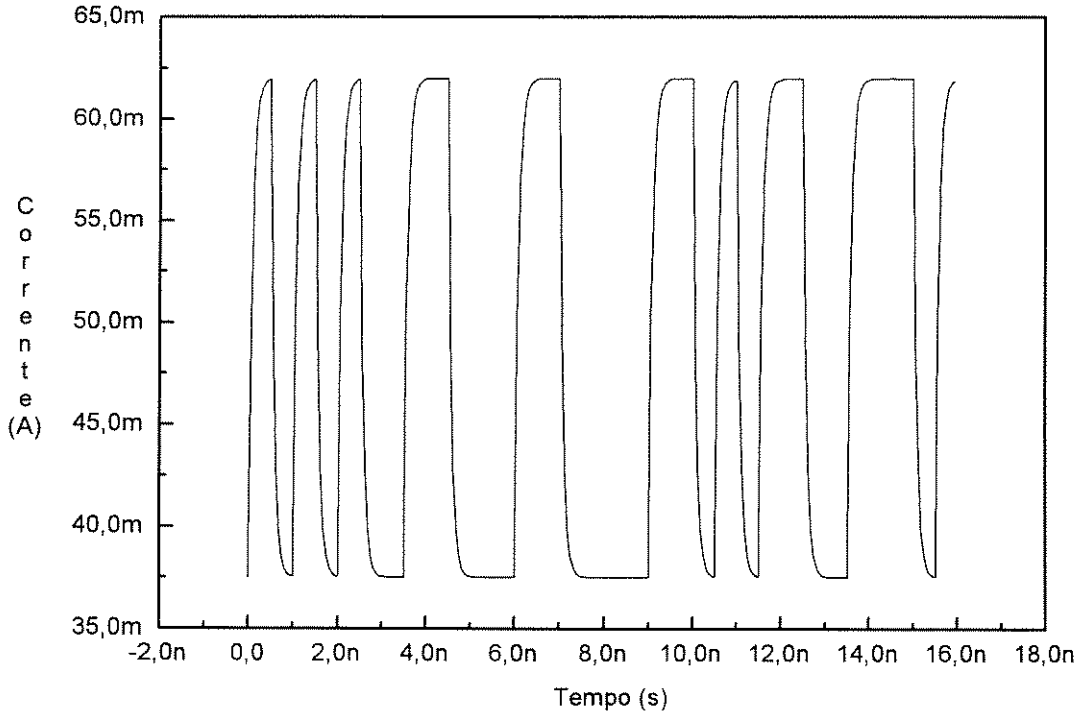


Figura 4.23 Corrente injetada em LASER2, 4, 6 e 8.

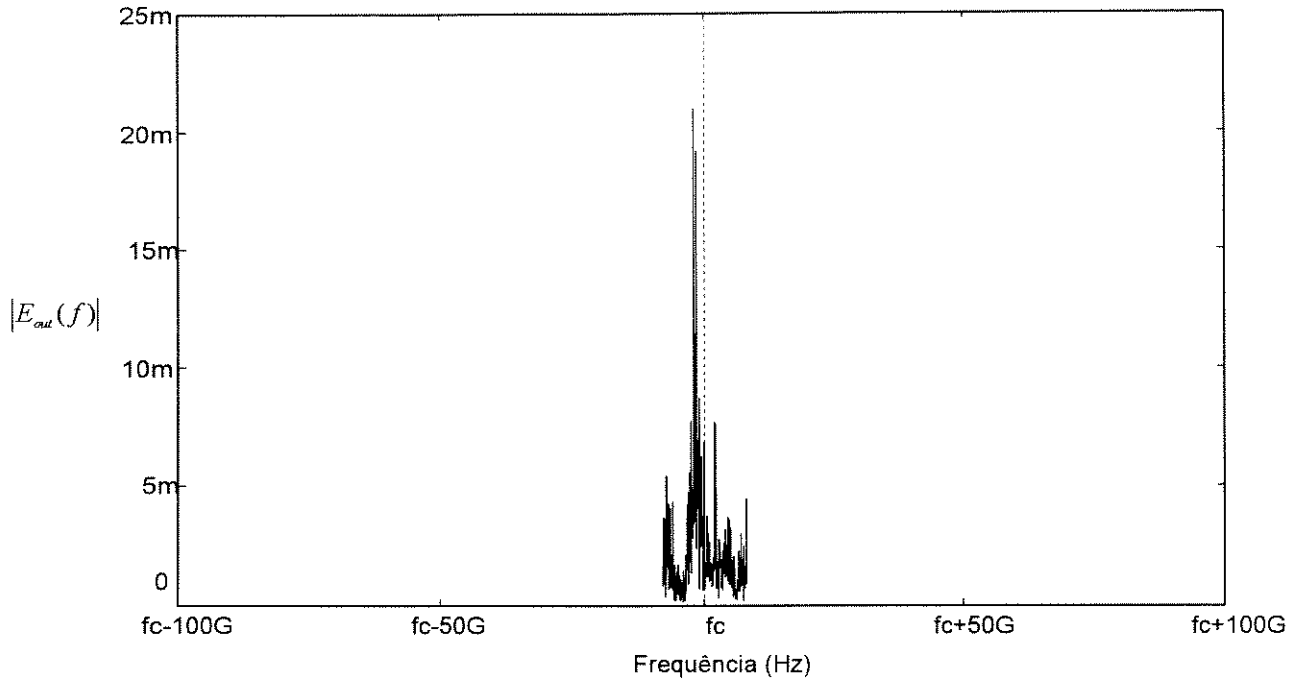


Figura 4.24 Espectro do campo elétrico na saída do LASER1,3,5 e 7.

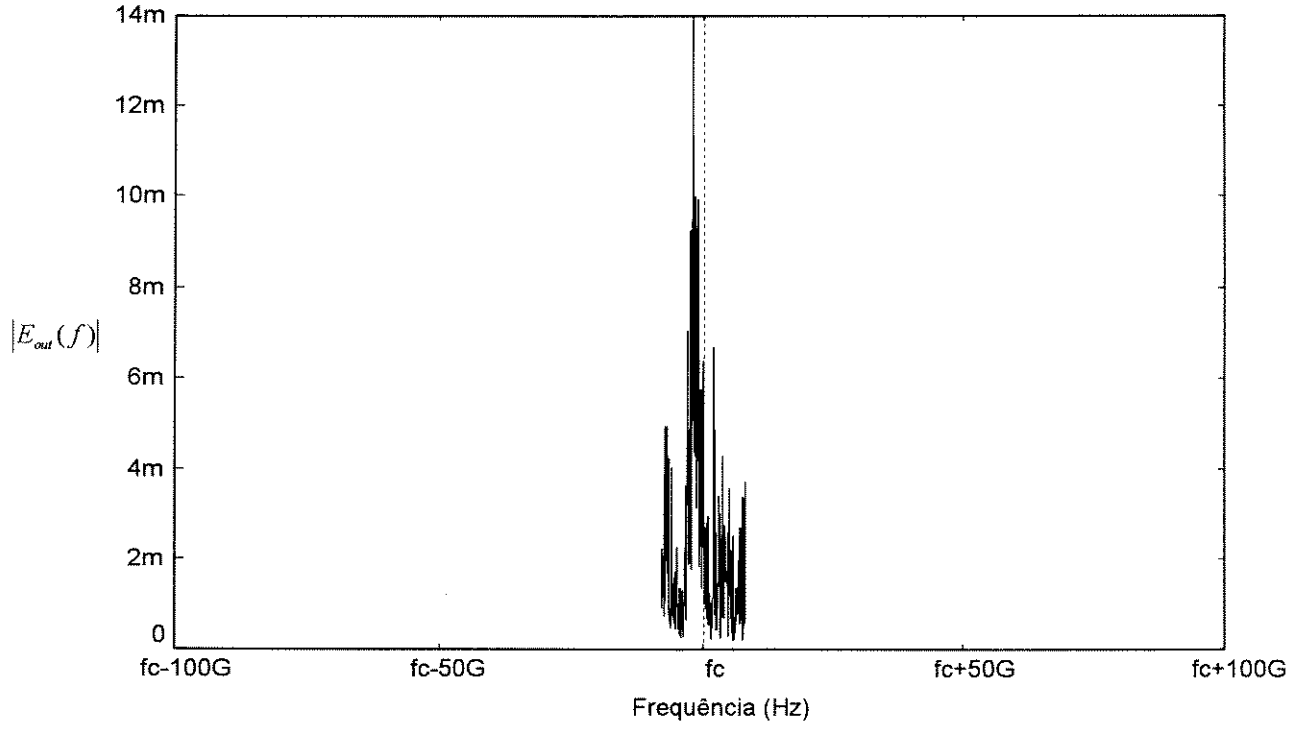


Figura 4.25 Espectro do campo elétrico na saída do LASER2,4,6 e 8.

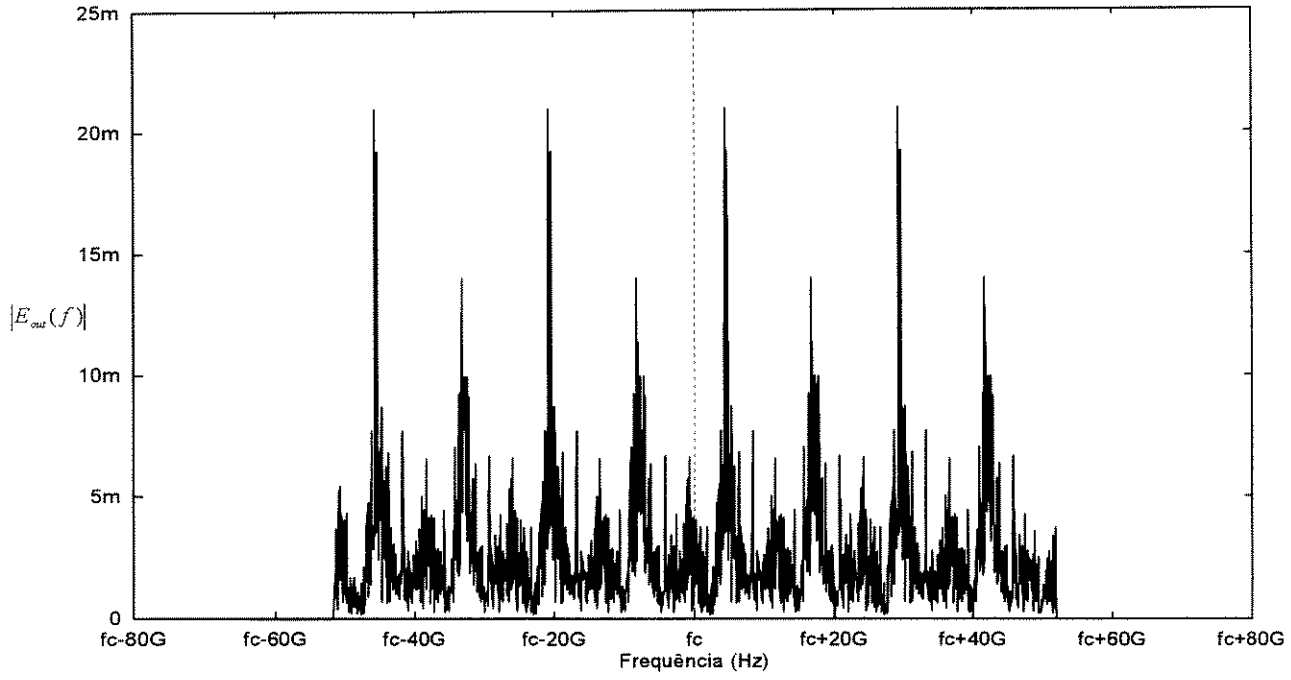


Figura 4.26 Espectro do campo elétrico na saída do ACOPLADOR, fc=193.542137097THz,espaçamento entre canais de 12GHz.

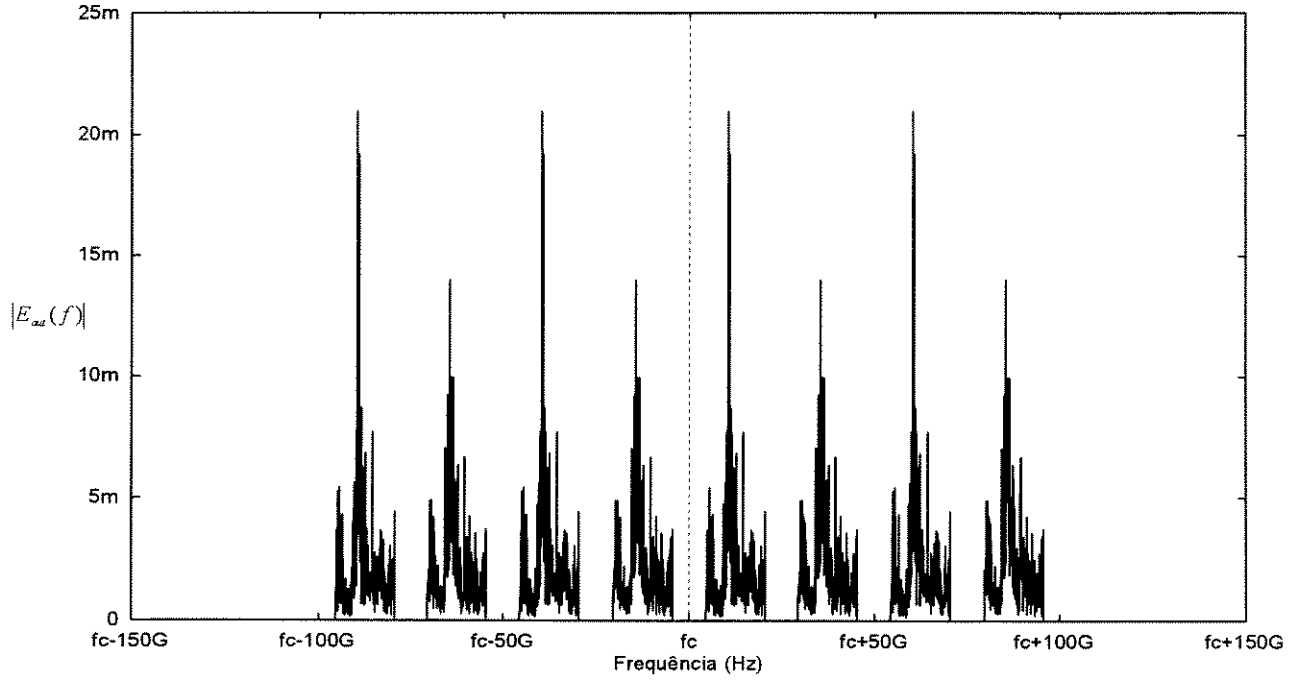


Figura 4.27 Espectro do campo elétrico na saída do ACOPLADOR, fc=193.535887097THz,espaçamento entre canais de 25GHz.

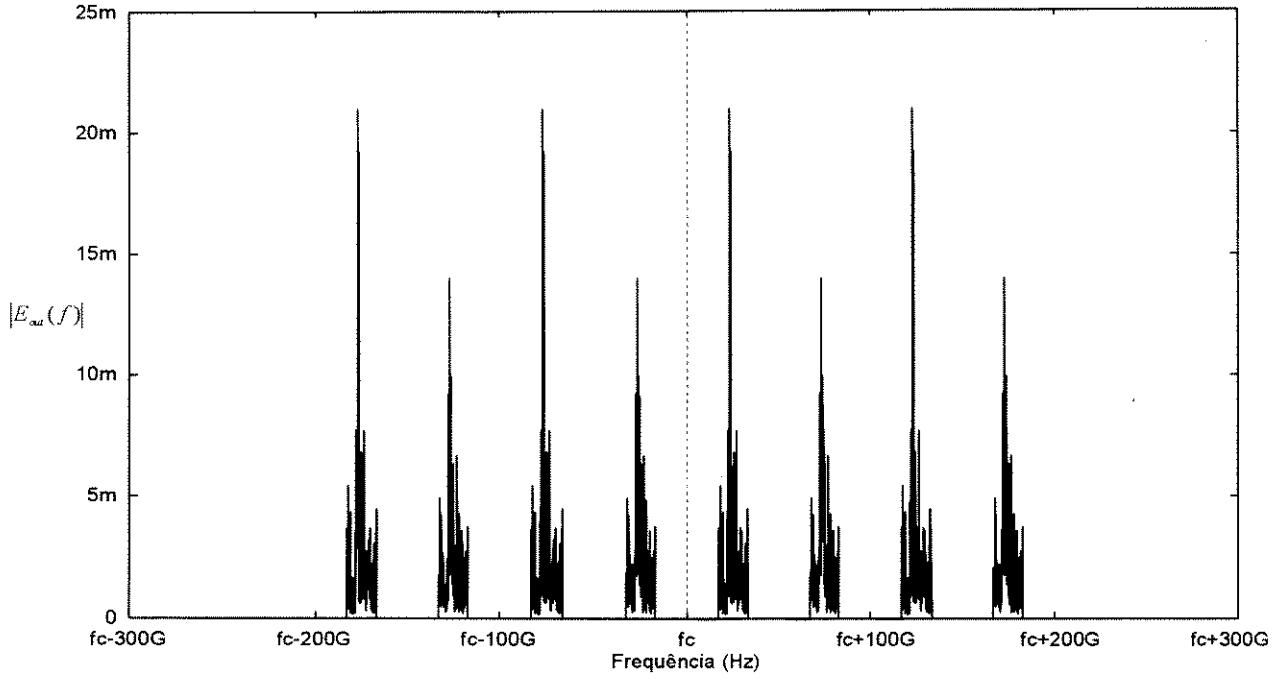


Figura 4.28 Espectro do campo elétrico na saída do ACOPLADOR, fc=193.523387097THz,espaçamento entre canais de 50GHz.

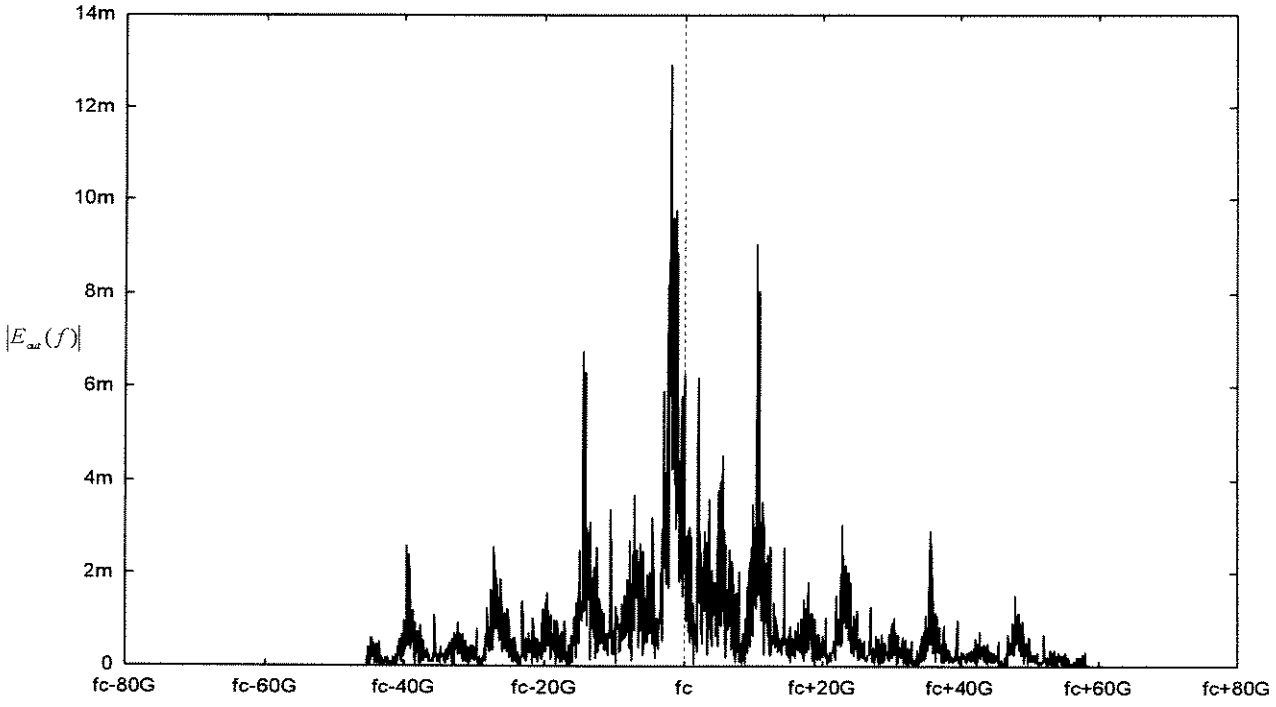


Figura 4.29 Espectro do campo elétrico na saída do FILTRO FP, fc=c/1.55μm, FWHM=10GHz, espaçamento entre canais de 12 GHz.

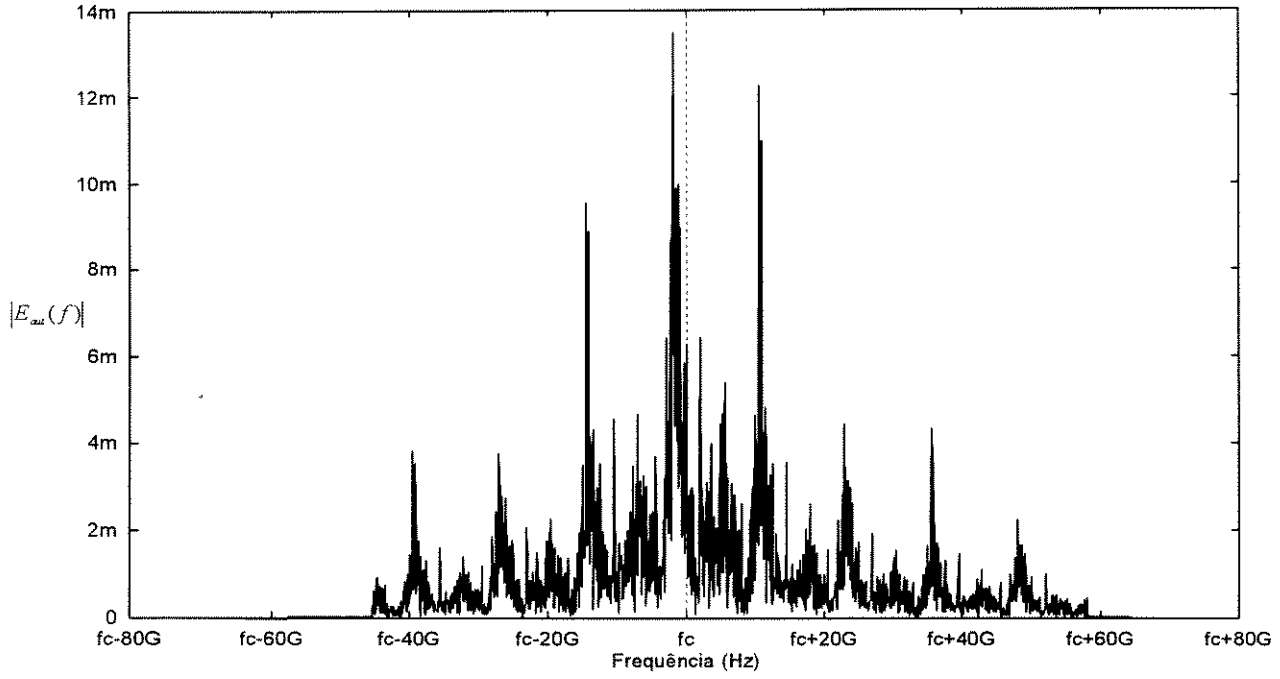


Figura 4.30 Espectro do campo elétrico na saída do FILTRO FP, fc=c/1.55μm, FWHM=15GHz, espaçamento entre canais de 12 GHz.

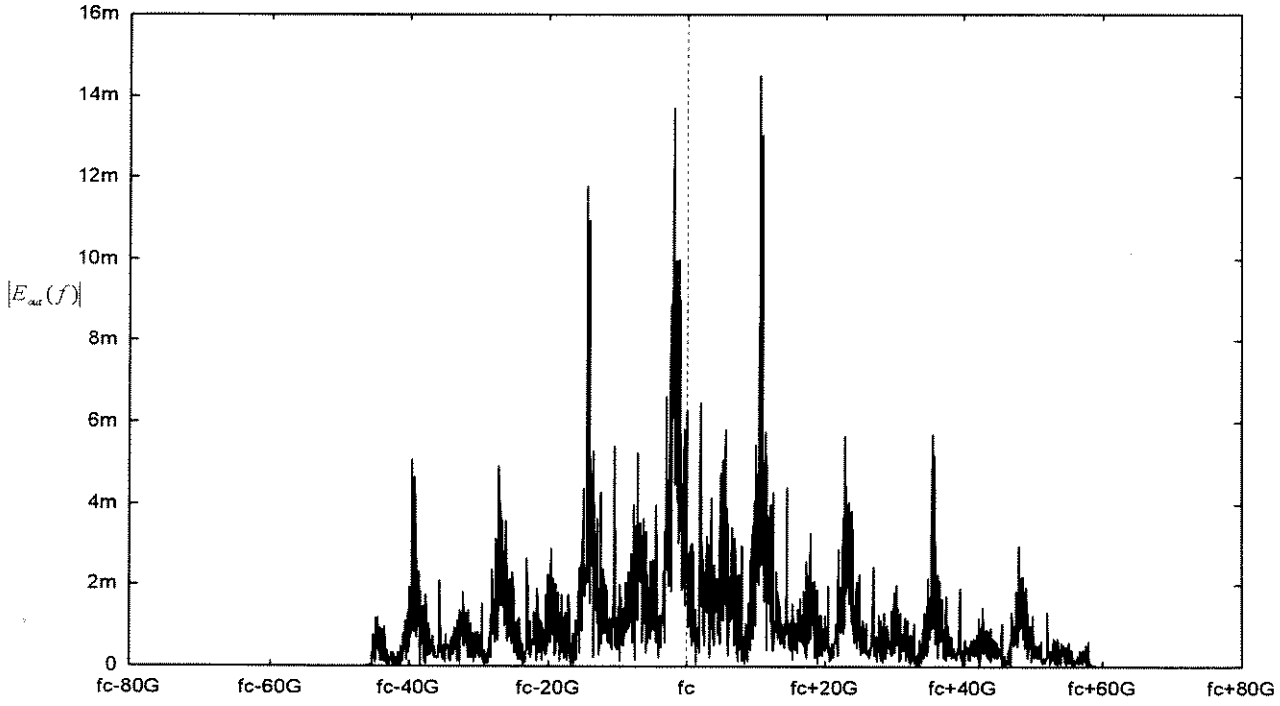


Figura 4.31 Espectro do campo elétrico na saída do FILTRO FP, fc=c/1.55μm, FWHM=20GHz, espaçamento entre canais de 12 GHz.

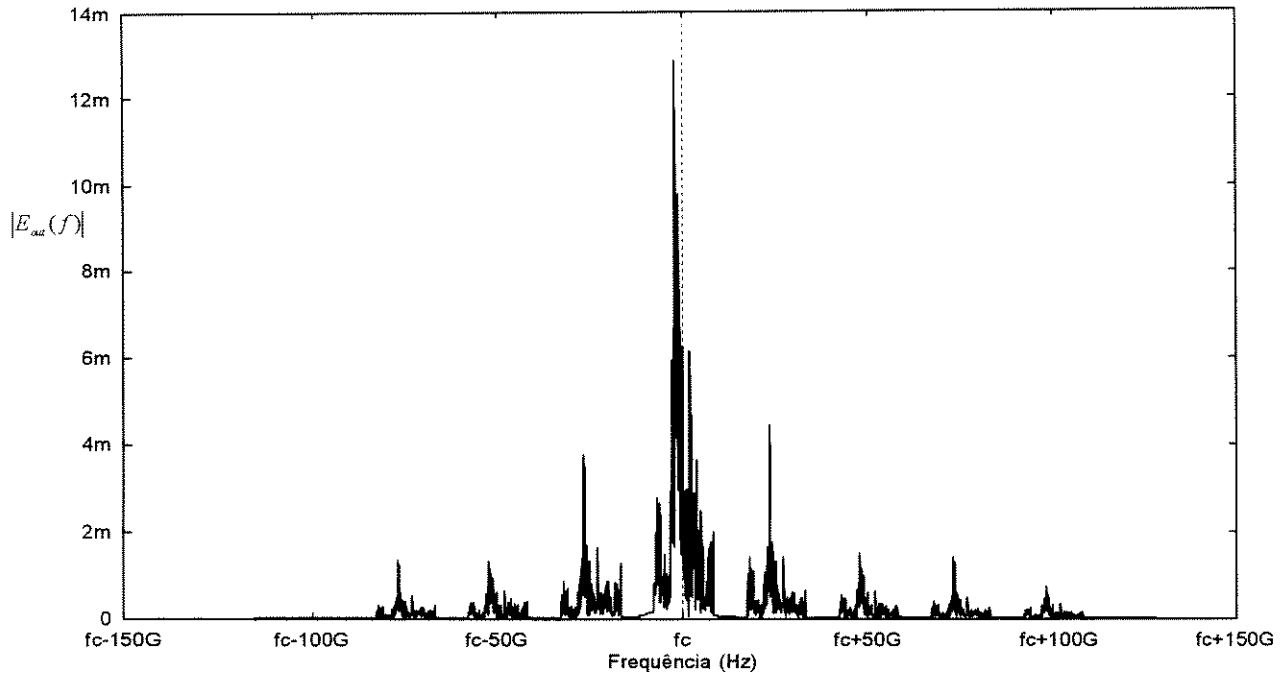


Figura 4.32 Espectro do campo elétrico na saída do FILTRO FP, fc=c/1.55μm, FWHM=10GHz, espaçamento entre canais de 25 GHz.

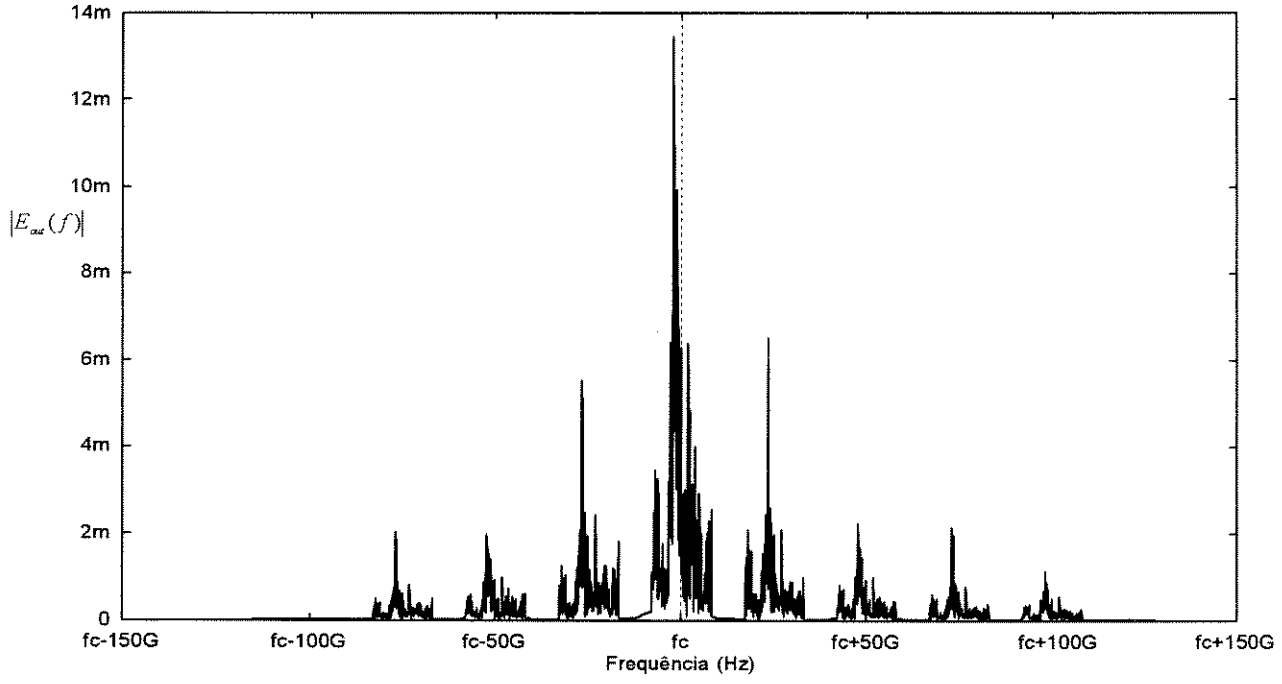


Figura 4.33 Espectro do campo elétrico na saída do FILTRO FP, fc=c/1.55μm, FWHM=15GHz, espaçamento entre canais de 25 GHz.

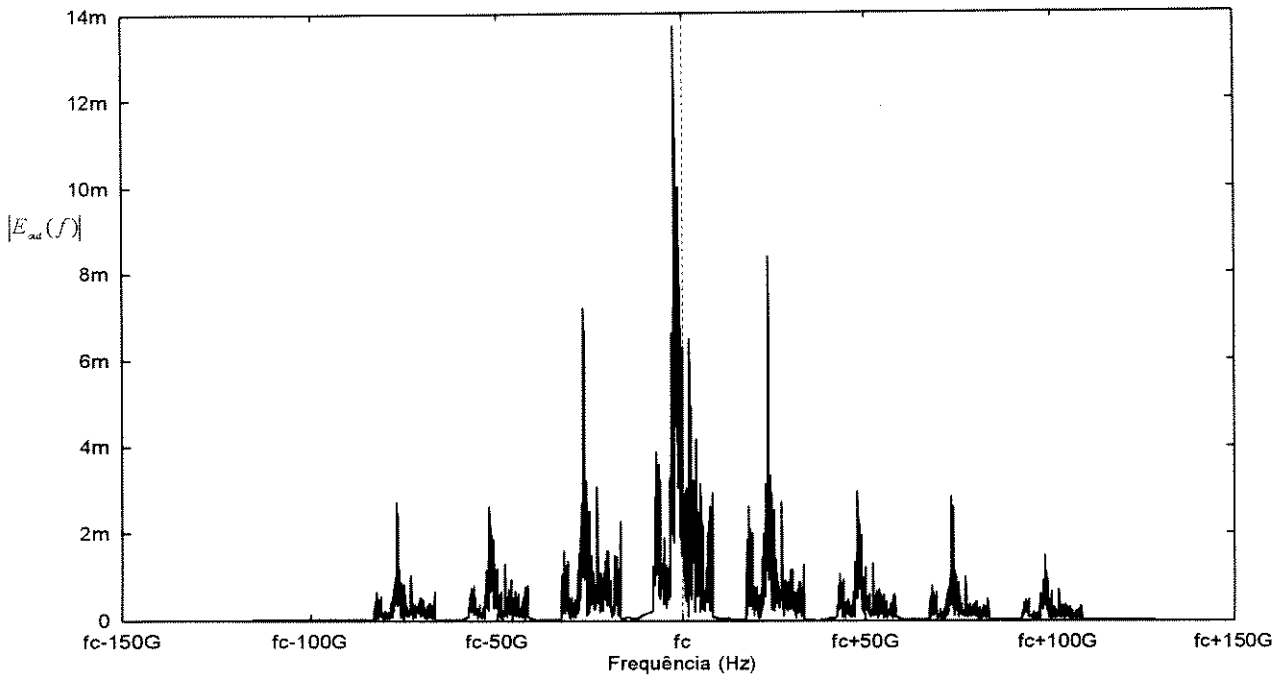


Figura 4.34 Espectro do campo elétrico na saída do FILTRO FP, fc=c/1.55μm, FWHM=20GHz, espaçamento entre canais de 25 GHz.

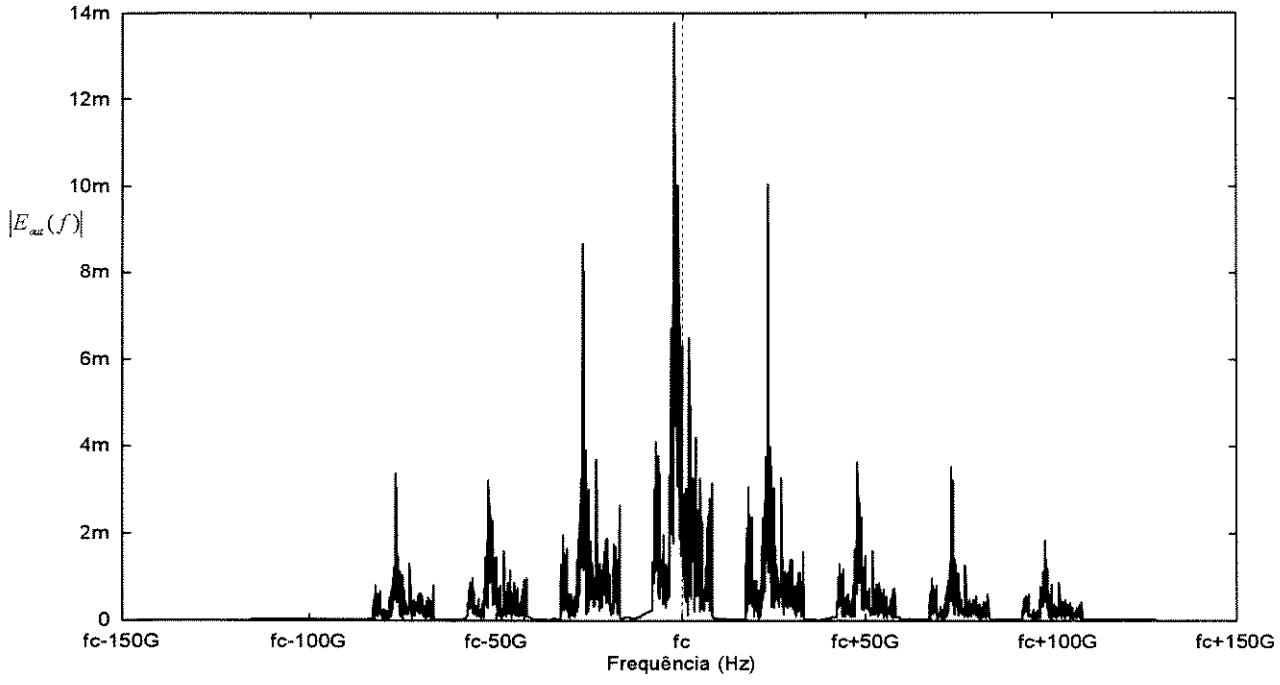


Figura 4.35 Espectro do campo elétrico na saída do FILTRO FP, fc=c/1.55μm, FWHM=25GHz, espaçamento entre canais de 25 GHz.

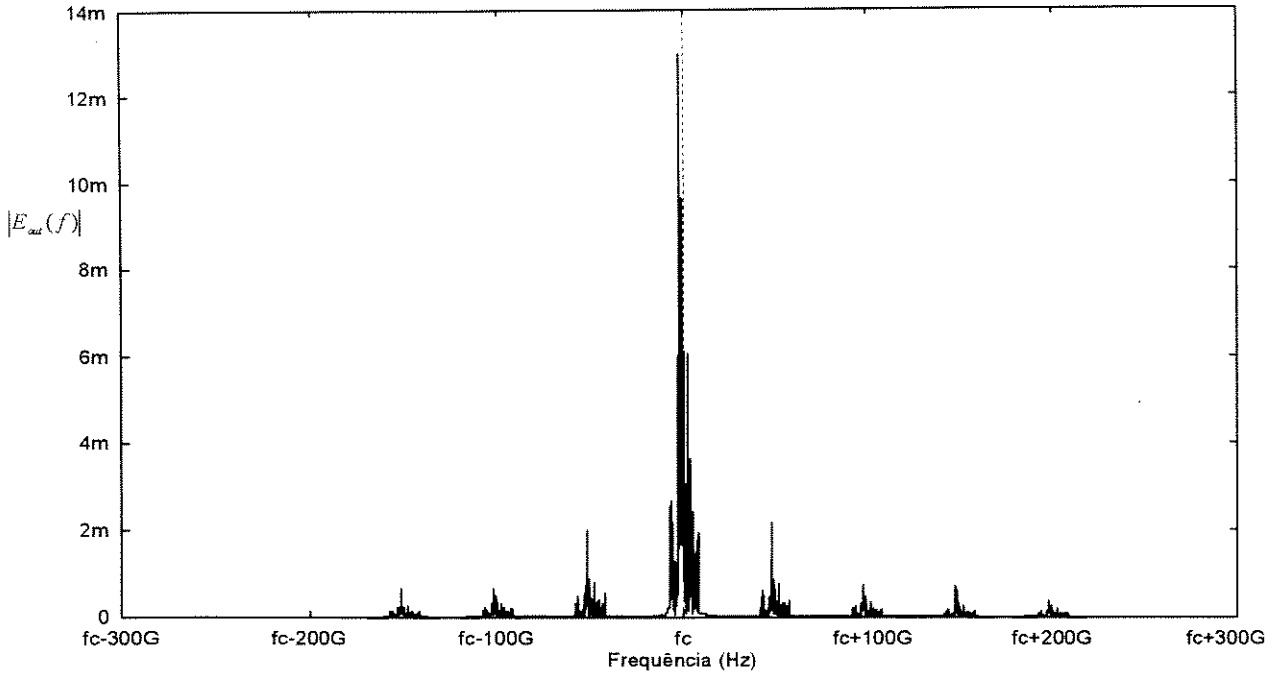


Figura 4.36 Espectro do campo elétrico na saída do FILTRO FP, fc=c/1.55μm, FWHM=10GHz, espaçamento entre canais de 50 GHz.

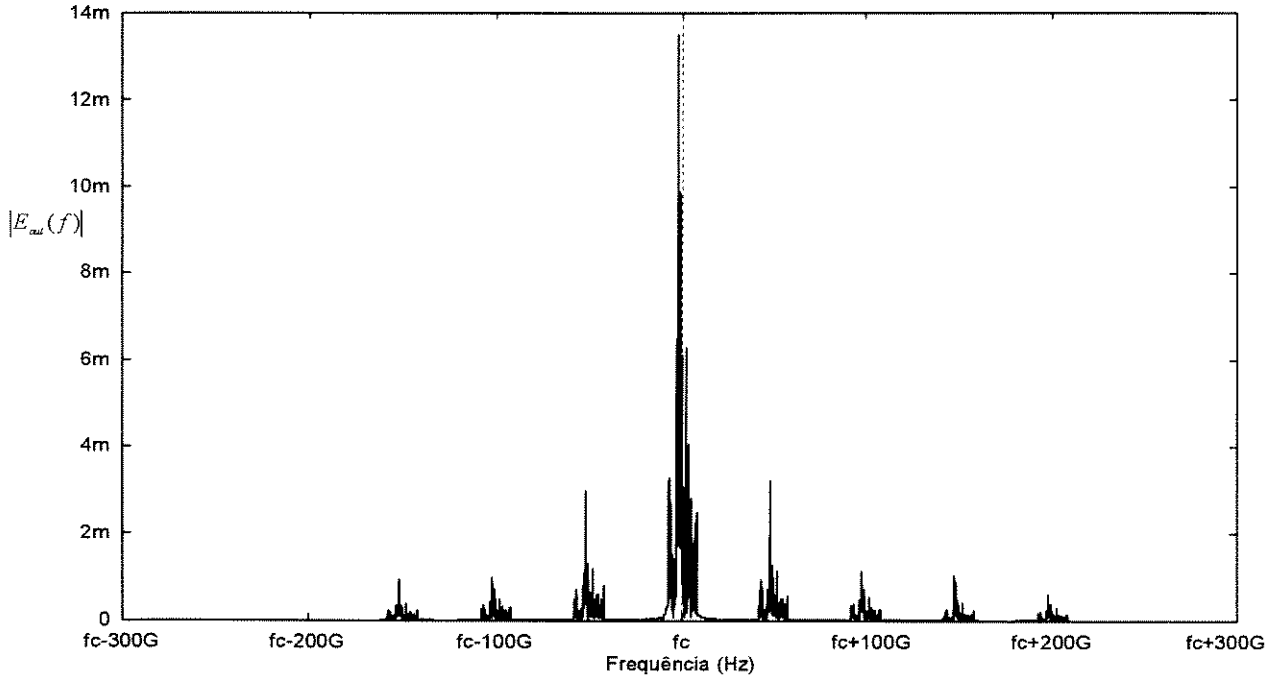


Figura 4.37 Espectro do campo elétrico na saída do FILTRO FP, fc=c/1.55μm, FWHM=15GHz, espaçamento entre canais de 50 GHz.

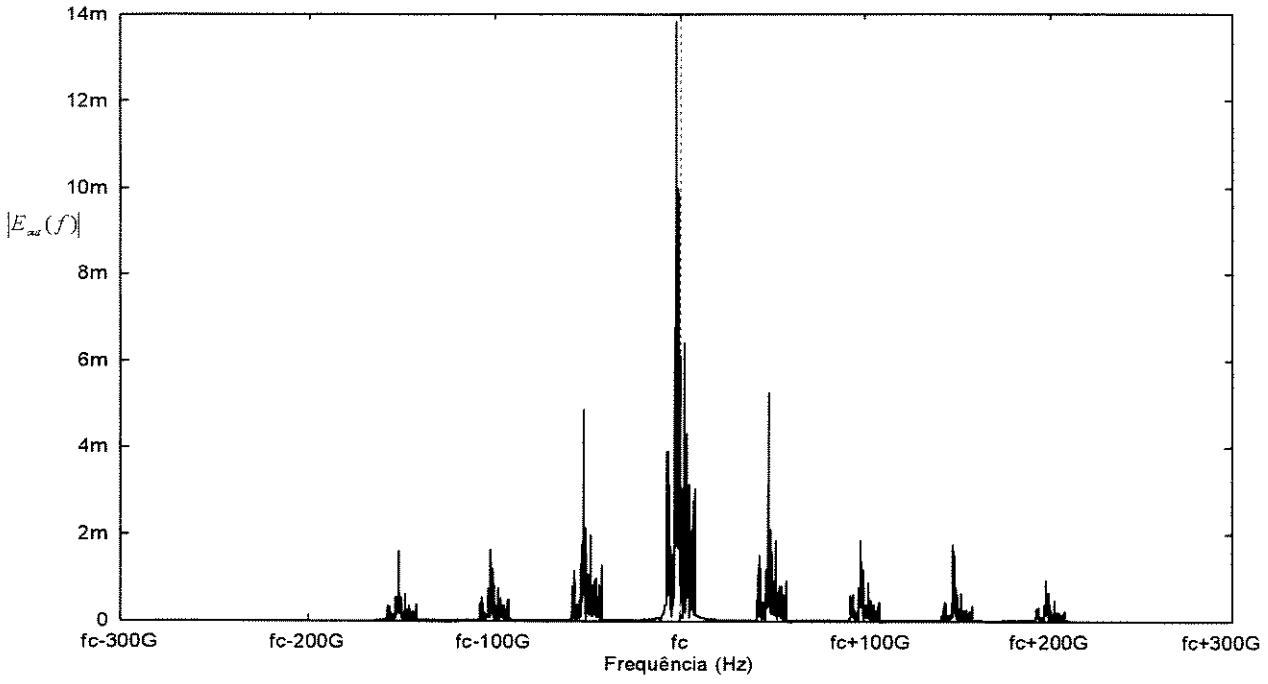


Figura 4.38 Espectro do campo elétrico na saída do FILTRO FP, fc=c/1.55μm, FWHM=25GHz, espaçamento entre canais de 50 GHz.

Para termos uma avaliação mais clara da interferência dos canais adjacentes, foram geradas as figuras 4.39-4.47, onde são mostrados os diagramas de olho na saída do filtro de formato gaussiano. O fechamento dos diagramas de olho permite uma avaliação comparativa do aumento da probabilidade de erro do sistema. As figuras 4.40 (FWHM=15GHz) e 4.41 (FWHM=20GHz) representam os casos mais críticos (espaçamento de 12 GHz), pois a contribuição dos canais adjacentes é muito pronunciada, entretanto, uma redução da banda do filtro de Fabry-Perot para 10 GHz minimiza bastante o problema (figura 4.39).

Quanto às figuras restantes, observamos que o maior espaçamento entre canais (25 e 50 GHz) permite uma maior tolerância em relação ao parâmetro FWHM. Na figura 4.44, por exemplo, o espaçamento e a banda do filtro tem o mesmo valor de 25 GHz.

Cabe aqui uma última observação, sistemas utilizando menores taxas de transmissão permitem uma maior proximidade entre os canais, devido a redução da banda de frequência dos lasers, um exemplo é a implementação em laboratório de um sistema WDM de 100 canais realizado pela NTT[2], operando em 622 Mb/s, onde o espaçamento entre canais é de 10 GHz, a sintonia dos canais utiliza um interferômetro de Mach-Zehnder.

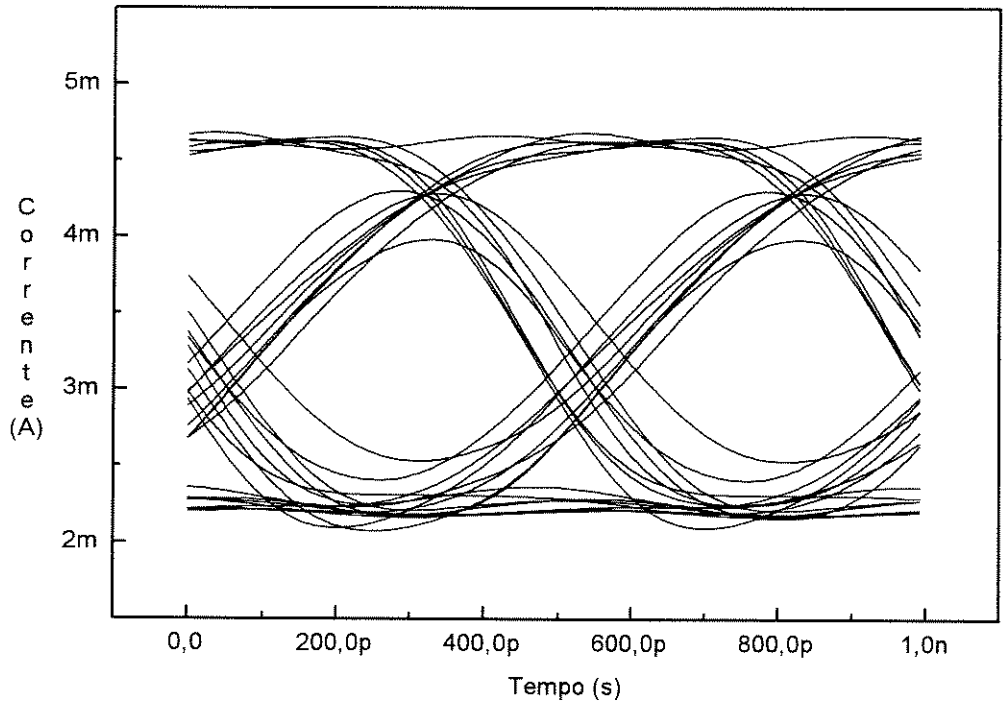


Figura 4.39 Diagrama de olho gerado com o sinal na saída do FILTRO, espaçamento entre canais de 12GHz, FWHM=10GHz.

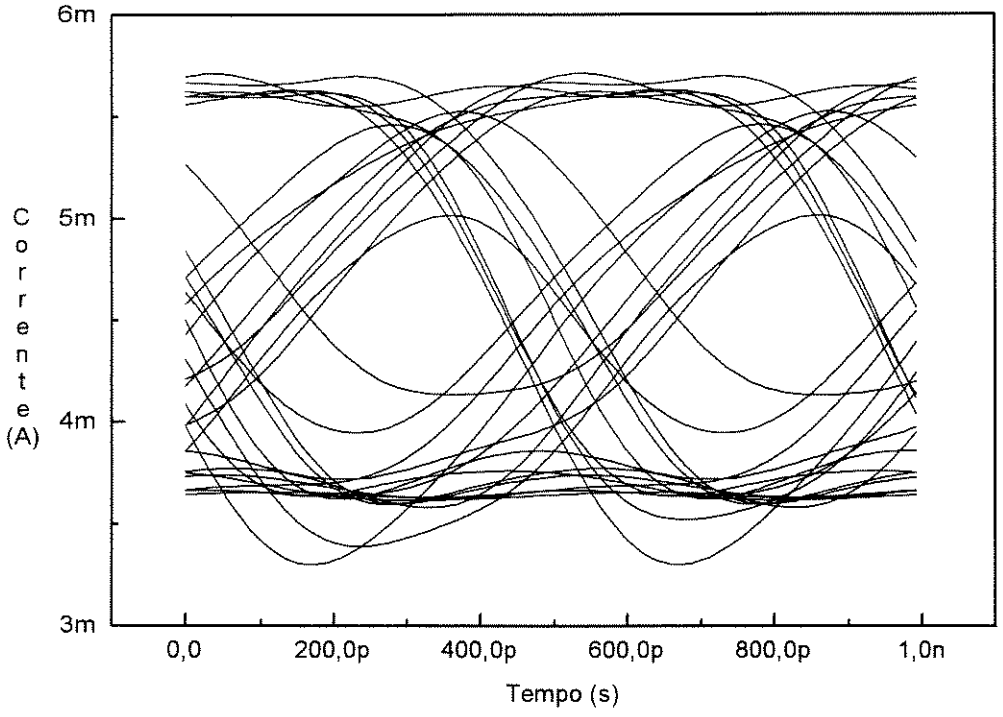


Figura 4.40 Diagrama de olho gerado com o sinal na saída do FILTRO, espaçamento entre canais de 12GHz, FWHM=15GHz.

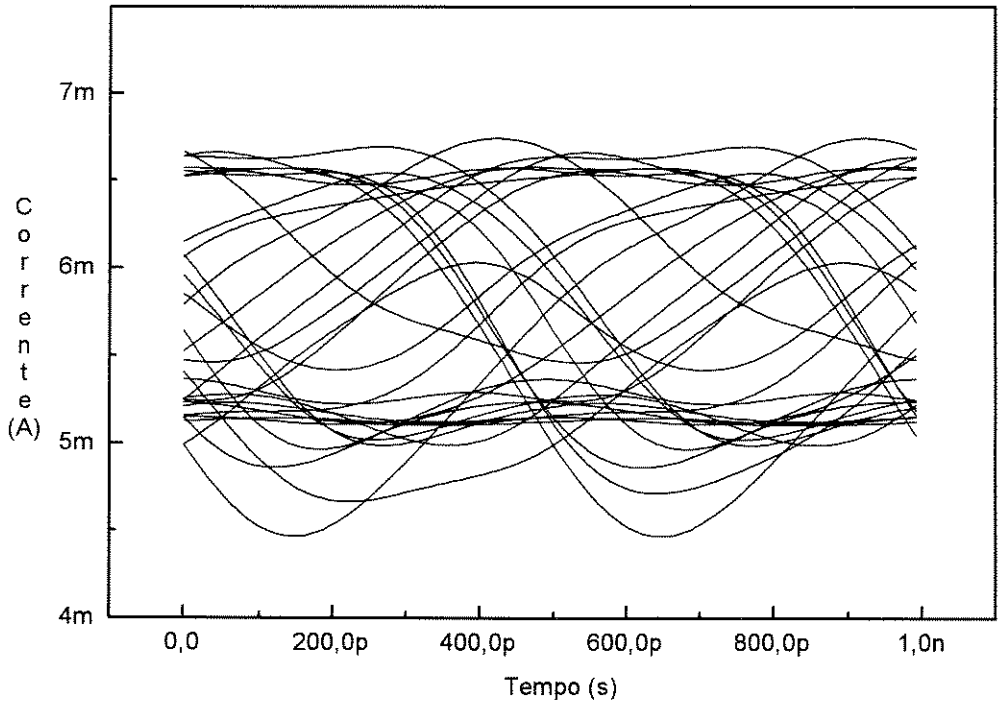


Figura 4.41 Diagrama de olho gerado com o sinal na saída do FILTRO, espaçamento entre canais de 12GHz, FWHM=20GHz.

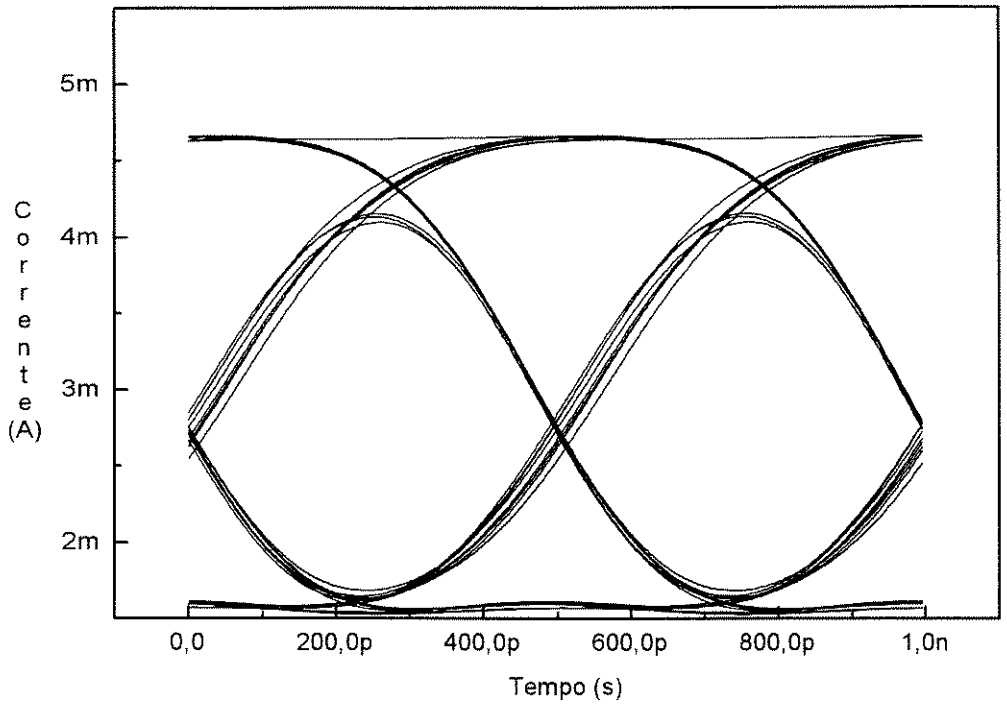


Figura 4.42 Diagrama de olho gerado com o sinal na saída do FILTRO, espaçamento entre canais de 25GHz, FWHM=15GHz.

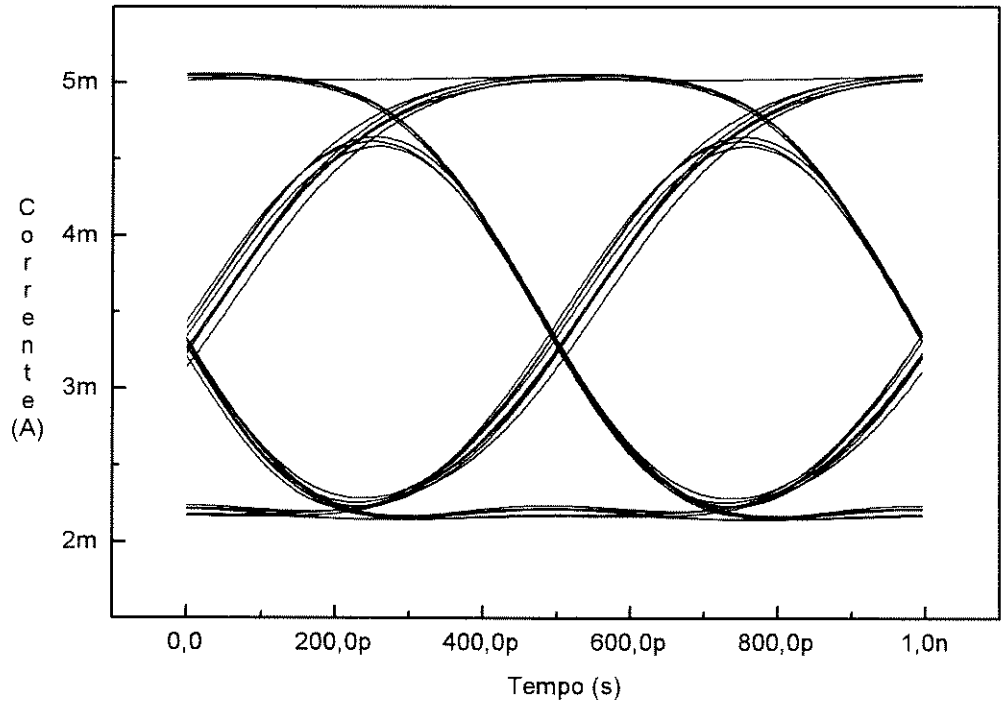


Figura 4.43 Diagrama de olho gerado com o sinal na saída do FILTRO, espaçamento entre canais de 25GHz, FWHM=20GHz.

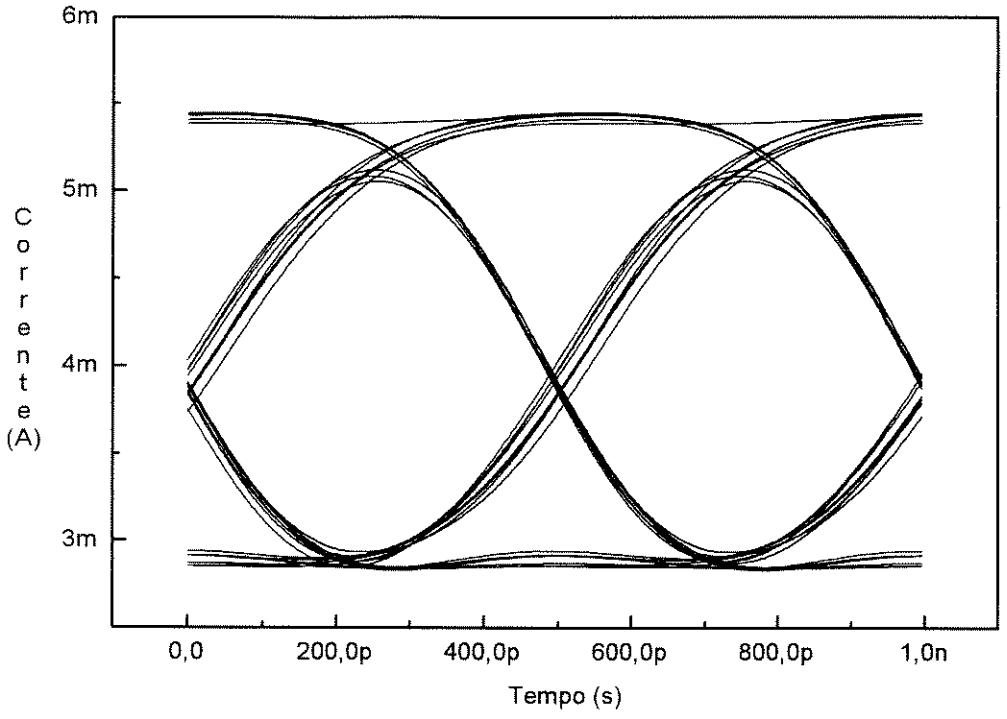


Figura 4.44 Diagrama de olho gerado com o sinal na saída do FILTRO, espaçamento entre canais de 25GHz, FWHM=25GHz.

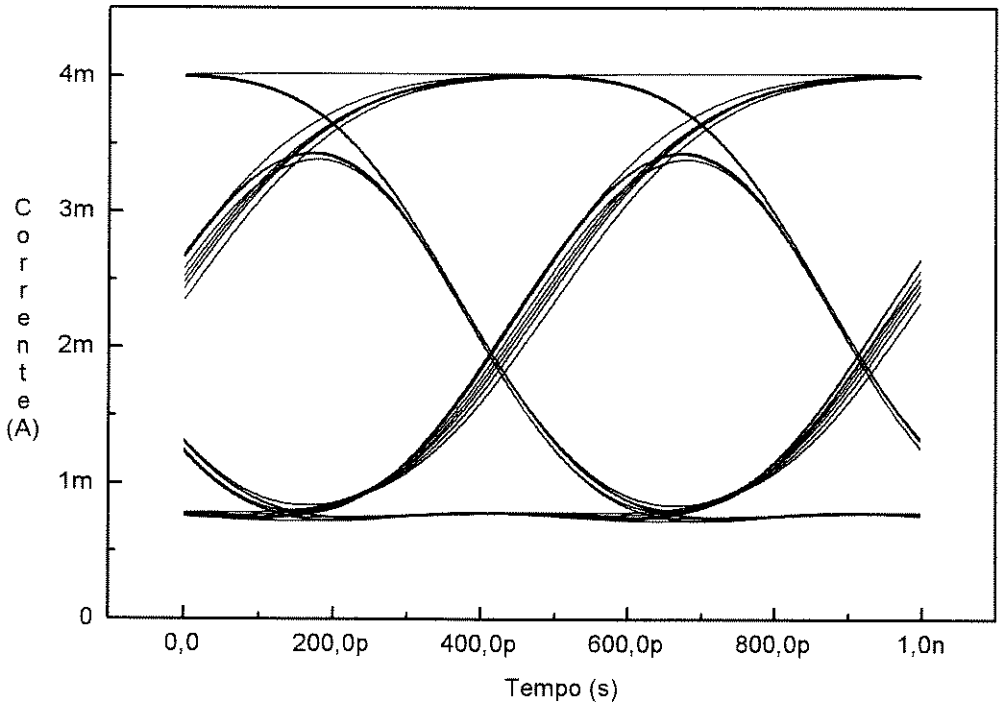


Figura 4.45 Diagrama de olho gerado com o sinal na saída do FILTRO, espaçamento entre canais de 50GHz, FWHM=10GHz.

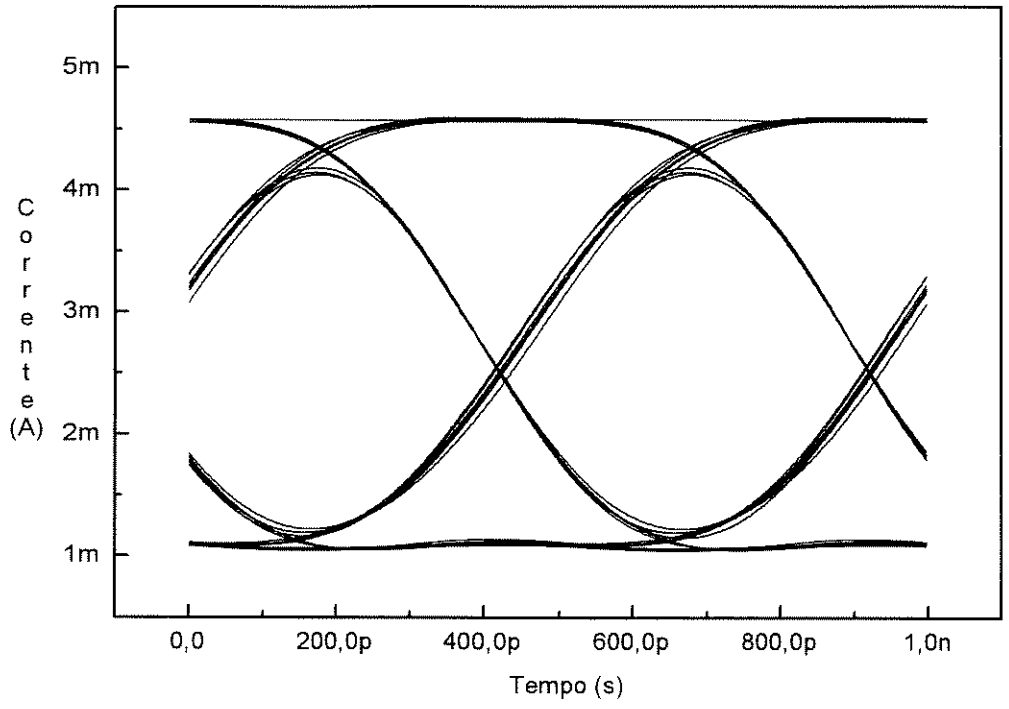


Figura 4.46 Diagrama de olho gerado com o sinal na saída do FILTRO, espaçamento entre canais de 50GHz, FWHM=20GHz.

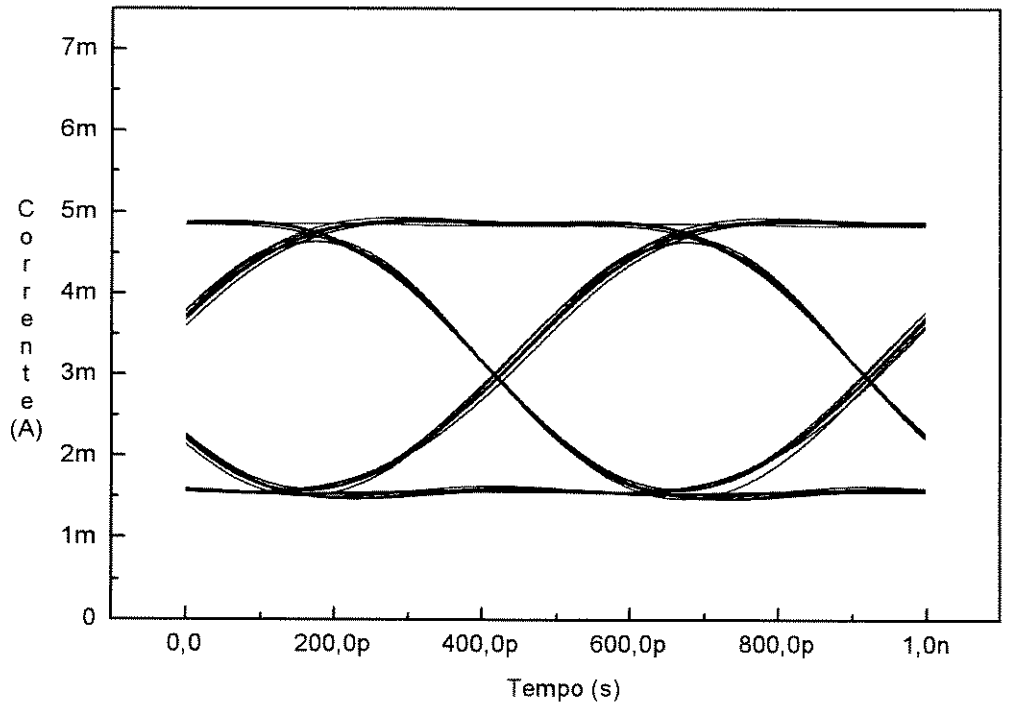


Figura 4.47 Diagrama de olho gerado com o sinal na saída do FILTRO, espaçamento entre canais de 50GHz, FWHM=30GHz.

4.6 Referências Bibliográficas

[1] G. Keiser, *Optical Fiber Communications*, McGraw-Hill, 1991.

[2] P. E. Green jr, *Fiber Optic Networks*, Prentice Hall, 1993.

[3] D. G. Messerschmitt, C. Li, F. F. Tong, K. Liu, "Channel Capacity Optimization of Chirp-Limited Dense WDM/WDMA Systems Using OOK/FSK Modulation and Optical Filters", *Journal of Lightwave Technology*, vol 10, No. 8, pp 1148-1161, agosto de 1992.

[4] J. C. Cartledge, G. S. Burley, "The Effect of Laser Chirping on Lightwave System Performance", *Journal of Lightwave Technology*, vol 7, No. 3, pp 568-573, março de 1989.

[5] P. J. Corvini, T. L. Koch, "Computer Simulation of Hight-Bit-Rate Optical Fiber Transmission Using Single-Frequency Lasers", *Journal of Lightwave Technology*, vol LT-5, No. 11, pp 1591-1595, novembro de 1987.

[6] K. Hinton, T. Stephens, "Modeling High-Speed Optical Transmission Systems", *IEEE Journal of Selected Areas in Communications*, vol 11, No. 3, abril de 1993.

[7] W. H. Press, B. P. Flannery, S. A. Teukolsky, W. T. Vetterling, *Numerical Recipes in C*, Cambridge University Press, 1991.

[8] H. A. Haus, *Waves and Fields in Optoeletronics*, Prentice-Hall, 1984.

[9] M. C. Jeruchim, "Techniques for Estimating the Bit Error Rate in the Simulation of Digital Communication Systems", *IEEE Journal of Selected Areas in Communications*, vol SAC-2, No. 1, janeiro de 1984.

Conclusão

Neste trabalho foi apresentado o SIMNT, um software de simulação de sistemas de comunicação usando técnicas modernas de simulação tais como: estrutura modular, topologia aberta, integração de dispositivos e sistemas, definição de blocos realimentados, biblioteca de modelos programável e ordenação e execução automáticas. O programa é desenvolvido em C++ no sistema operacional Windows NT™.

Foi apresentada a estrutura de construção do programa, mostrando as estruturas de classes que formam o programa. Esta estrutura de classes permite um entendimento natural do funcionamento do SIMNT. Neste trabalho, apresentou-se também uma descrição do funcionamento do programa através da simulação de um sistema particular idealizado de forma didática. Finalmente, foram apresentados os resultados da simulação de três sistemas ópticos usando o SIMNT: dois enlaces ópticos e um sistema WDM. Estes sistemas foram escolhidos por apresentarem arquiteturas diversas, com o objetivo de demostrar os recursos do programa.

Os resultados gerados durante a simulação dos sistemas ópticos permitiram uma avaliação de alguns fatores limitantes na transmissão de dados a altas taxas e longas distâncias. Os dois primeiros enlaces ilustraram o efeito da dispersão cromática da fibra monomodo aliado ao *chirp* do laser, causando um fechamento do diagrama de olho com o aumento da distância de transmissão. O sistema WDM permitiu avaliarmos a interferência entre canais, através de diagramas de olho e espectro do campo elétrico para diversos espaçamentos entre canais e larguras de banda do filtro de Fabry-Perot.

Durante a utilização do SIMNT, observamos que o tempo de processamento ocupado pelo executor, responsável pelo gerenciamento da simulação, foi desprezível.

Os próximos objetivos do trabalho são: desenvolvimento de uma interface gráfica amigável; utilização dos recursos de multienlace do WINDOWS NT™, diminuindo o tempo de processamento numérico; implementação de uma biblioteca de modelos para simulação de sistemas ópticos, integrando as atividades de modelamento de dispositivos ópticos do DECOM; desenvolvimento das rotinas de pós-processamento numérico, inclusive métodos de cálculo de probabilidade de erro, como o método de Monte-Carlo.

Além do DECOM, este programa esta sendo testado no departamento de engenharia elétrica da Universidade Federal do Espírito Santo, sob a orientação dos professores Marcelo E. Vieira Segatto e Moisés R. Nunes Ribeiro.

Apêndice A

Funções de Interface para Desenvolvimento dos Modelos

A.1 Introdução

Descreveremos neste apêndice as macros e funções de interface disponíveis para implementação de modelos para a classe *BlockLib*. Estas funções podem ser divididas em três grupos: macros para manipulação de parâmetros, funções de entrada e funções de saída.

A.2 Macros para Manipulação de Parâmetros

A.2.1 READPARAM

A.2.1.1 Sintaxe

READPARAM(tipo var);

A.2.1.2 Descrição

Esta macro retorna em *var* o valor que se encontra no arquivo de parâmetros associado ao bloco. *var* deve ter sido declarada com o mesmo nome e *tipo (int, int*, long, long* , char*, char**, double, double*, complex ou complex**) que foi declarada no arquivo de parâmetros. No caso de vetores, deve ser alocada a memória necessária antes da chamada a READPARAM.

A.2.1.3 Exemplo

Carrega o parâmetro *Tamanho* e o vetor *Sequencia*:

```
int Tamanho;
READPARAM(Tamanho);
double *Sequencia=new double[Tamanho];
READPARAM(Sequencia);
```

Arquivo de parâmetros:

```
int Tamanho 10 # tamanho do vetor
double Sequencia 1.2 1.4 1.6 1.7 8.7 5.7 0.7 0.5 4.3 5.6 # sequência de valores
```

A.2.2 LOAD e SAVE

A.2.2.1 Sintaxe-

```
LOAD(tipo var);
.
.
.
SAVE(tipo var);
```

A.2.2.2 Descrição

São usadas em conjunto, na primeira chamada ao bloco, LOAD informa ao SIMNT que *var* deve ser armazenada como variável de estado do bloco, SAVE atualiza seu valor. Nas próximas chamadas ao bloco, LOAD retorna o valor anteriormente gravado por SAVE, e assim sucessivamente. Estas macros substituem a declaração *static*[1]. *var* deve ser declarada antes da chamada à LOAD, e admite os mesmos tipos que READPARAM.

A.2.2.3 Exemplo

```
int Contador=0;
LOAD(Contador);
//na primeira chamada, Contador=0, segunda chamada, Contador=1.
Contador++;
SAVE(Contador);
```

A.3 Funções de Leitura

A.3.1 ReadSize

A.3.1.1 Sintaxe

long ReadSize(int in);

A.3.1.2 Descrição

Retorna o número de amostras disponíveis na entrada *in* do bloco.

A.3.1.3 Exemplo

```
long Tamanho;
Tamanho=ReadSize(0);
```

A.3.2 ReadMax

A.3.2.1 Sintaxe

long ReadMax(int out);

A.3.2.2 Descrição

Retorna o número máximo de amostras definido no arquivo de topologia para a saída *out*.

A.3.2.3 Exemplo

```
long Max;
Max=ReadMax(0);
```

A.3.3 ReadLabel

A.3.3.1 Sintaxe

char ReadLabel(int in);*

A.3.3.2 Descrição

Retorna um ponteiro para o tipo *char* com o comentário na entrada *in.*

A.3.3.3 Exemplo

```
char *Comentario;
Comentario=ReadLabel(0);
```

A.3.4 ReadDomain

A.3.4.1 Sintaxe

int ReadDomain(int in);

A.3.4.2 Descrição

Retorna um inteiro informado se as amostras da entrada *in* estão no domínio do tempo (*int TIME=0*) ou frequência (*int FREQ=1*).

A.3.4.3 Exemplo

```
int Dom=ReadDomain(0);
if(Dom==FREQ)
        cerr<<"Erro 109 : entrada 0 deve estar no domínio do tempo"<<endl;
```

A.3.5 ReadType

A.3.5.1 Sintaxe

int ReadType(int in);

A.3.5.2 Descrição

Retorna o tipo (*DOUBLE=0, COMPLEX=1 ou INT=2*) das amostras na entrada *in*.

A.3.5.3 Exemplo

```
complex *x;
int Tipo=ReadType(0);
if(Tipo==COMPLEX) x=new complex[100];
```

A.3.6 ReadSampleFreq

A.3.6.1 Sintaxe

```
double ReadSampleFreq(int in);
```

A.3.6.2 Descrição

Retorna a frequência de amostragem das amostras na entrada *in*.

A.3.6.3 Exemplo

```
double Fs;
Fs=ReadSample(0);
```

A.3.7 ReadCenterFreq

A.3.7.1 Sintaxe

```
double ReadCenterFreq(int in);
```

A.3.7.2 Descrição

Retorna a frequência central das amostras na entrada *in*.

A.3.7.3 Exemplo

```
double Fc;
Fc=ReadCenterFreq(0);
```

A.3.8 ReadInitTime

A.3.8.1 Sintaxe

```
double ReadInitTime(int in);
```

A.3.8.2 Descrição

Retorna o valor do instante de tempo da primeira amostra disponível na entrada *in*.

A.3.8.3 Exemplo

```
double Time;
Time=ReadInitTime(0);
```

A.3.9 ReadTime

A.3.9.1 Sintaxe

```
double ReadTime(int in,long n);
```

A.3.9.2 Descrição

Retorna o instante de tempo associado a amostra *n*, na entrada *in*.

A.3.9.3 Exemplo

```
long Tam=ReadSize(0);
long n=Tam/2;
double Time=ReadTime(0,n);
```

A.3.10 ReadSample

A.3.10.1 Sintaxe

*void * ReadSample(int in, long size);*

A.3.10.2 Descrição

Retorna um vetor de amostras de tamanho *size* na entrada *in*. É necessária a conversão explícita do tipo *void* para os tipos *double*, *complex* ou *int*.

A.3.10.3 Exemplo

```
double* Double;
complex *Complex;
long Tamanho=ReadSize(0);
int Tipo=ReadType(0);
if(Tipo==COMPLEX) Complex=(complex*) ReadSample(0,Tamanho);
else if(Tipo==DOUBLE) Double=(double*) ReadSample(0,Tamanho);
```

A.3.11 WriteDelay

A.3.11.1 Sintaxe

void WriteDelay(int in, long delay);

A.3.11.2 Descrição

Introduz um atraso de *delay* amostras na entrada *in*.

A.3.11.3 Exemplo

```
long Delay;
READPARAM(Delay);
WriteDelay(0,Delay);
```

A.4 Funções de Escrita

A.4.1 WriteStatus

A.4.1.1 Sintaxe

void WriteStatus(int status);

A.4.1.2 Descrição

Informa o estado *status* do bloco ao SIMNT. Um bloco pode ter três estados: DEAD, indicando que o bloco terminou seu processamento; STOP, indicando que o bloco terminou seu processamento e não pode ser chamado novamente, pois não dispõe de espaço para armazenar mais amostras; AGAIN, informando que deve ser chamado novamente.

A.4.1.3 Exemplo

```
long Tamanho=ReadSize(0);
long Max=ReadMax(0);
// se não dispõe de amostras na entrada zero informa ao SIMNT que não pode
// fazer nada
if(Tamanho==0) WriteStatus(DEAD);
// se o número de amostras na entrada zero é maior do que sua capacidade de
// armazenamento na saída zero , informa ao SIMNT que não pode mais ser
// chamado.
else if(Tamanho>MAX) WriteStatus(STOP);
// tudo ok, informa que deve ser chamado novamente
else WriteStatus(AGAIN);
```

A.4.2 WriteProp

A.4.2.1 Sintaxe

void WriteProp(int out, int tipo, int dominio,int unidade, double amostragem, double FreqCentral, double Tempo, char comentario);*

A.4.2.2 Descrição

Registra todas as propriedades do vetor de amostras que será colocado na saída *out* (tipo, domínio,frequência de amostragem, frequência central, instante inicial de tempo e um comentário).

A.4.2.3 Exemplo

```
// escreve na saída zero as propriedades da entrada um
WriteProp(0, ReadType(1), ReadDomain(1), 0, ReadSampleFreq(1),
ReadCenterFreq(1), ReadInitTime(1), "novo comentário");
```

A.4.3 WriteSample

A.4.3.1 Sintaxe

```
int WriteSample(int out, tipo* vetor, long tam);
```

A.4.3.2 Descrição

Escreve na saída *out* o vetor de amostras de tamanho *tam*, onde o vetor pode ser de três tipos: *double, complex* e *int*. Se o valor de retorno for OVERF, indicando que o buffer de saída esta cheio, o bloco deverá informar ao SIMNT que seu estado é STOP.

A.4.3.3 Exemplo

```
double *x=new double[1000];
WriteProp(0, DOUBLE, TIME, 0, 10, 100, 0, "comentário");

if(WriteSample(0,x,1000)==OVERF) WriteStatus(STOP)
else WriteStatus(DEAD);
```