UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

FÁBIO BECK WANDERER

Implementação de rádio definido por software em FPGA

Porto Alegre

2016

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

FÁBIO BECK WANDERER

Implementação de rádio definido por software em FPGA

Projeto de Diplomação apresentado ao Departamento de Engenharia Elétrica da Escola de Engenharia da Universidade Federal do Rio Grande do Sul, como requisito parcial para Graduação em Engenharia Elétrica

Orientador: Prof. Dr. Hamilton Klimach

Porto Alegre
2016

CIP - Catalogação na Publicação

Beck Wanderer, Fábio
Implementação de rádio definido por software em FPGA / Fábio Beck Wanderer. -- 2016.
142 f.

Orientador: Hamilton Klimach.

Trabalho de conclusão de curso (Graduação) -- Universidade Federal do Rio Grande do Sul, Escola de Engenharia, Curso de Engenharia Elétrica, Porto Alegre, BR-RS, 2016.

1. Rádio definidido por software. 2. LabVIEW. 3. FPGA. I. Klimach, Hamilton, orient. II. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da UFRGS com os dados fornecidos pelo(a) autor(a).

FÁBIO BECK WANDERER

Implementação de rádio definido por software em FPGA

Projeto de Diplomação apresentado ao Departamento de Engenharia Elétrica da Escola de Engenharia da Universidade Federal do Rio Grande do Sul, como requisito parcial para Graduação em Engenharia Elétrica

---

Prof. Dr. Hamilton Klimach
Orientador - UFRGS

Prof. Dr. Ály Ferreira Flores Filho
Chefe do Departamento de Engenharia Elétrica (DELET) - UFRGS

Aprovado em 14 de dezembro de 2016.

BANCA EXAMINADORA

Prof. Dr. Hamilton Klimach
UFRGS

Prof. Dr. Giovani Bulla
UFRGS

Prof. Ms. Sérgio Luiz Schubert Severo
IFSul-Pelotas

A meus pais Luiz e Marla

Agradecimentos

À minha família. Em especial aos meus pais, por sempre me presentearam com a melhor estrutura para o alcance dos meus sonhos e objetivos.

À Gyorgya. Pelo companheirismo, paciência e apoio renovadores ao longo deste momento único.

A meus amigos. Em especial ao Dario, pelo apoio e presença incondicionais durante a necessidade de resiliências.

Aos meus colegas de curso Braian, Derek, Gabriel, Nibele, Ramede e Otávio. Sou imensamente grato pela presença e unidade vividas ao longo de toda nossa jornada. Sem vocês não teria chegado aonde cheguei.

Ao mestre Severo por vivenciar sua encorajente e inspiradora arte de ensinar. Ao professor Hamilton pela confiança na realização deste trabalho. Aos mestres Fabio e Lang por serem dignos das posições que brilhantemente ocupam. Ao mestre Serafini pela a confiança e cordialidade para o compartilhamento da experiência e conhecimento adquiridos.

"What I cannot create,
I do not understand"
(Richard Feynman)

Resumo

Este trabalho implementa um rádio definido por *software* através da plataforma heterogênea LabVIEW FPGA com um sistema operacional em tempo real. Serão apresentados os conceitos básicos sobre comunicação digital e seus pontos mais importantes para a transmissão e recepção de um sinal de aúdio analógico, uma descrição dos elementos de programação em LabVIEW empregados no desenvolvimento, uma descrição da plataforma, as características dos rádios definidos por *software*, a arquitetura do sistema desenvolvido e a descrição detalhada das rotinas. Ao final, serão apresentados os resultados obtidos com a arquitetura de *software* proposta com base em refências citadas ao longo do desenvolvimento do estudo, assimo como sugestões e melhorias para a continuação deste trabalho.

Palavras-chave: rádio definido por *software*. LabVIEW. FPGA. RT. comunicação digital. QAM.

Abstract

This work implements a software defined radio using the LabVIEW FPGA platform with a real-time operational system. Basic concepts regarding digital communication and its keys points to transmit and receive an analog audio signal will be presented. All the LabVIEW programming functions and routines in the development will be described alongside with the platform. Basics caracterists of software defined radios are mentioned. The achieved results will be presented ainthe end based on the proposed architecture and all the references regarding this study as well as suggestions for improvments and continuation of this work.

Keywords: LabVIEW. FPGA. RTOS. digital communications. QAM.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

ADC	*Analog to Digital Converter*
AI	*Analog Input*
AO	*Analog Output*
AM	*Amplitude Modulation*
BD	*Block Diagram*
BPSK	*Binary Phase-Shift Keying*
BRAM	*Block Random Access Memory*
CLB	*Configurable Logic Block*
DAC	*Digital to Analog Converter*
DMA	*Direct Memory Access*
DSP	*Digital Signal Processing*
EVM	*Error Vector Magnitude*
FFT	*Fast Fourier Transform*
FIFO	*First In First Out*
FIR	*Finite Inpulse Response*
FM	*Frequency Modulation*
FP	*Front Panel*
FPGA	*Field Programmable Gate Array*
I8	*8 bit Integer*
I16	*16 bit Integer*
IF	*Intermediary Frequency*
I/O	*Input/Output*
IQ	*In-phase Signal Quadrature Signal*

ISI	*Inter Symbol Interference*
LabVIEW	*Laboratory Virtual Instrument Engineering Workbench*
LE	*Logic Element*
LED	*Light-emitting Diode*
LUT	*Lookup Table*
MER	*Modulation Error Ratio*
MUX	*Multiplexer*
NI	*National Instruments*
QAM	*Quadrature Amplitude Modulation*
RF	*Radio Frequency*
RT	*Real Time*
RT FIFO	*Real Time FIFO*
RTOS	*Real Time Operational System*
SDR	*Software Defined Radio*
SCTL	*Single-cycle Timed Loop*
SoC	*Sytem on Chip*
STFT	*Short Time Fourier Transform*
U32	*32 bit Unsigned*
USB	*Universal Serial Bus*
VI	*Virtual Instrument*
VHDL	*VHSIC Hardware Description Language*
VHSIC	*Very High Speed Integrated Circuit*
U32	*32 bit Unsigned*

Sumário

Introdução

De acordo com a (SDR FORUM, 2007) um SDR (*Software Defined Radio*) é um rádio em que algumas ou todas as funções das camadas físicas podem ser definidas por *software*. As camadas físicas, eletrônica implementada em *hardware* como a de um filtro analógico, por exemplo, serão realizadas em software. Assim, diferentemente do rádio tradicional analógico, agora todas as partes condicionantes ao sinal para ser transmitido, como filtros, misturadores e moduladores podem ser feitos em ambiente digital, através de *software*.

Este tema é abordado formalmente em (MITOLA, 1992) onde é observado que através de uma utilização eficiente de recursos computacionais é possível tornar esta arquitetura possível tanto para um usuário quanto para um fabricante dos componentes eletrônicos a serem utilizados. Com uma rápida evolução dos dispositivos semicondutores dedicados ao processamento de sinais digitas, os DSPs (*Digital Signal Processing*), pode-se utilizar cada vez menos componentes analógicos em conjunto com o SDR. A parte menos robusta para constituir um rádio totalmente definido por *software* é a que utiliza os ADC's ou DAC's pois nestes estão amarradas as taxas de amostragem e conversão que podem realizar e uma vez operando com sinal de frequências na faixa de GHz, suas performances podem não ser tão eficientes dependendo de sua construção. Assim, em alguns casos se opta por utilizar algum um outro bloco analógico para a conversão da frequência da portadora para uma frequência intermediária, do inglês *intermediary frequency* (IF), e assim posteriormente um ADC que opere com esta IF.

Neste trabalho será apresentado o conceito de rádio definido por *softwate*, sua teoria e equações que o definem. Será mostrada a plataforma de desenvilvimento para sua implementação, assim como a descrição da mesma. Abordar-se-ão conceitos importantes do ambiente de programação LabVIEW, como boas práticas e estratégias para implementação de rotinas específicas usadas no desenvolvimento deste trabalho. Mencionar-se-á um breve conceito de um sistema operacional em tempo real, assim como a caracterização do FGPA presente no placa de desenvolvimento myRIO.

A implementação proposta tem como modelo a proposta por (KEHTARNAVAZ; MAHOTRA, 2010). As modificações e adaptações estão presentes nas seções 3 e 4. Os resultados obtidos pela implementação sugerida encontram-se no capítulo 5.

1 Objetivos

Na busca por um melhor entendimento desta tecnologia recente dispõe-se a construir um SDR utilizando a plataforma heterogênea NI RIO, isto é, utilizar a combinação entre os dois módulos específicos, RT e FPGA para avaliar sua adequação, potencial e limites. Propor uma arquitetura e desenvolver um *software* em LabVIEW buscando o máximo desempenho possível em taxa de transmissão. Para isto utilizar-se-á o produto NI myRIO. Assim também, identificar, na arquitetura proposta, os pontos onde se é possível buscar por melhorias e propor possíveis desenvolvimentos para atingi-los.

2 Plataforma de desenvolvimento

A partir da escolha da linguagem de programação utilizada pelo *software* LabVIEW, serão analisadas diretrizes sugeridas pela documentação de sua empresa desenvolvedora, *National Instruments*. Como serão utilizados dois ambientes de execução, FPGA e o sistema operacional de tempo real (RTOS), primeiramente serão apresentados conceitos importantes destas duas instâncias.

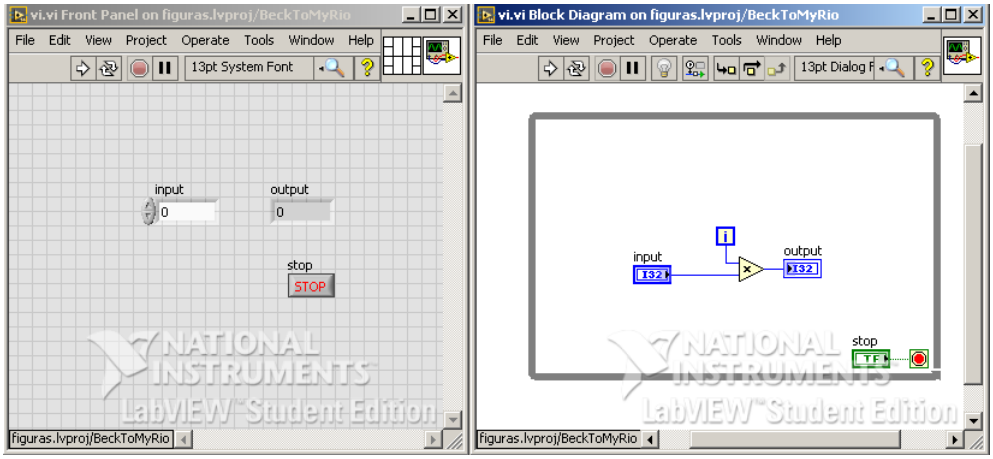
2.1 LabVIEW

Laboratory Virtual Instrument Engineering Workbench (LabVIEW) é um ambiente de desenvolvimento integrado com base em uma programação gráfica desenvolvido pela empresa *National Instruments* (NI). A execução do código é feita de acordo com a estrutura gráfica do diagrama de blocos, (BD - *block diagram*), seguindo um paradigma da programação por fluxo de dados - *dataflow programming*. As estruturas utilizadas pelo programador são conectadas através de “fios” e estes propagam as informações entre “nós” (*nodes*). A estrutura conectada ao "nó"seguinte executará no momento em que o dado estiver válido, ou seja, tendo chegado a este último ponto.

Ao se programar em ambiente LabVIEW o programador possui três componentes: um diagrama de blocos, uma interface com o usuário, o painel frontal (FP - *front panel*), e um painel de conexão (*connector panel*). No diagrama de blocos se encontra o código da aplicação em desenvolvimento, onde se encontram as estruturas e funções que realizam as operações com os controles, dados de entrada, etc. As rotinas de programação são chamadas de Instrumentos Virtuais (VI - *Virtual Instrument*) e podem ser compostas por sub-rotinas que são, por sua vez, também VI’s. Assim, dados de entradas e dados de saída podem ser visualizados no FP onde suas conexões, relações entre entrada e saída, podem ser visualizados no BD.

Uma das vantagens de se programar utilizando esta linguagem é a facilidade em se visualizar a estrutura de programa, uma vez que a linguagem é gráfica. É de rápido aprendizado e muito versátil para se aprender algoritmos de programação. Com as ferramentas para depuração, (*debugging*), é possível acompanhar cada parte do código e observar a mudança das variáveis, assim como o fluxo dos dados ao longo do tempo. No entanto, para rotinas mais elaboradas de programação torna-se necessário conhecer em mais detalhes a maneira com que as funções e estruturas de programação típicas desta linguagem funcionam. É necessário ter um maior aprofundamento sobre suas estruturas e como se comunicam com outras plataformas externas e outros dispositivos externos a um computador.

Figura 1 – FP à esquerda e BD à direita



Fonte: Produzido pelo autor

2.2 myRIO

O Labview tanto pode ser executado em um computador pessoal como ser utilizado em sistemas embarcados. A NI fabrica inúmeros dispositivos com a tecnologia desses sistemas. Entre eles o myRIO. É um dispositivo de hardware que possui entradas e saídas digitais e analógicas, entradas e saídas de áudio, botão externo, LED's e uma porta USB. Possui também um sistema operacional em tempo real, do inglês *real time operational system* (RT) ou (RTOS), e um FPGA (NATIONAL INSTRUMENTS, 2013).

2.3 *Real Time*

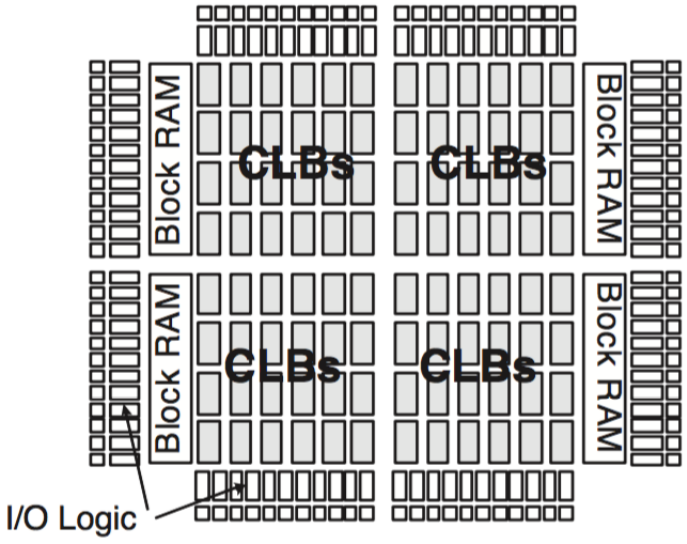
Internamente ao myRIO está embarcado em sua estrutura um RTOS. A diferença básica entre um *software* convencional e um de tempo real é o papel dominante das restrições temporais (SHAW, 2001). Um sistema determinístico é aquele que necessita ocorrer em um intervalo de tempo muito bem definido. Caso o sistema não opere no intervalo desejado, pode haver um erro de temporização, ou ainda não ser constante. Esta diferença entre o tempo desejado e o tempo que pode variar é chamado de *jitter*. Para sistema RT há um controle muito maior para estas condições, pois estes são dedicados a apenas executarem uma única rotina de programação. Com o LabVIEW RT é possível ter acesso a este tipo de sistema. Sistemas operacionais de tempo real são importantes para aplicações de engenharia onde certas tarefas necessitam ocorrer sempre em uma mesmo intervalo de tempo ou com a mesma frequência sem que hajam grandes variações. Quando se utiliza um sistema operacional, *operational system* (OS), de uso genérico, ou seja, não dedicado especificamente para alguma tarefa, não se pode garantir que a instrução seja

realizada em um tempo conhecido. Aliado a este módulo existe também o módulo de FPGA do LabVIEW.

2.4 FPGA

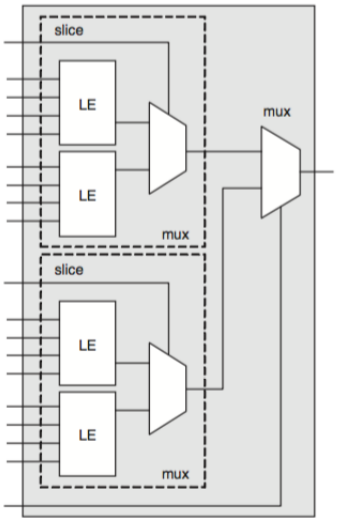
"A ideia principal do FPGA é de se utilizar lógica programável para a implementação de simples funções lógicas" (WIŚNIEWSKI, 2009). FPGA's oferecem uma grande vantagem para sistemas embarcados que necessitam de paralelismo. Estes possuem três componentes: blocos lógicos configuráveis, *configurable logic blocks* - CLB's, entradas e saídas, *input/output* (I/O) e blocos de memórias dedicados, *dedicated memory blocks* - BRAM's, Figura 2. Todos elementos são conectados através de interconexões programáveis. Nos CLB's estão presentes os elementos lógicos, *logic elements* - LE's, e multiplexadores *MUX*. Cada CLB é composto por 4 LE's, onde 2 LE's constituem um *slice*, Figura 3. Os elementos básicos de um LE são uma *lookup table* (LUT) com 4 entradas e uma saída, um MUX e um *flip-flog*, Figura 4.

Figura 2 – Estrutura típica de um FPGA



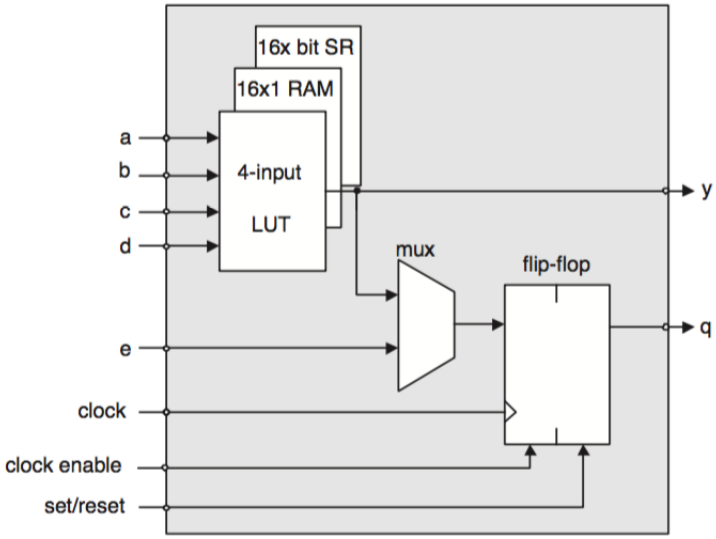
Fonte: (WIŚNIEWSKI, 2009)

Figura 3 – Estrutura de um CLB



Fonte: (WIŚNIEWSKI, 2009)

Figura 4 – Estrutura típica de um LE



Fonte: (WIŚNIEWSKI, 2009)

Comparados aos processadores, os FPGA's possuem um clock de execução mais baixo, no entanto, permitem, através de seu paralelismop que distintas porções do FPGA realizem diferentes funções em um único ciclo de *clock*. Para embarcar o código no FPGA, o LabVIEW FPGA primeiro converte o código escrito em sua linguagem para linguagem em descrição de *hardware* VHSIC (*very high speed integrated circuit*), VHDL (*VHSIC*

hardware description language), o envia ao compilador (Xilinx) que o sintetiza em um arquivo *bitfile* e este é embarcado no *hardware* e opera conforme instruções projetadas no ambiente LabVIEW (NATIONAL INSTRUMENTS, 2014).

2.5 Recursos de programação no LabVIEW RT/FPGA

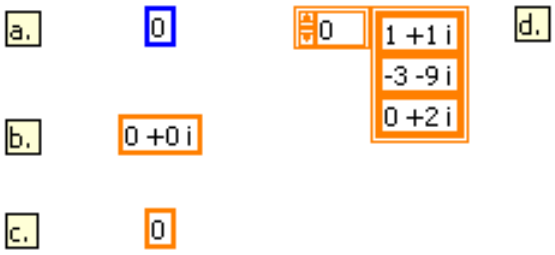
Para uma melhor utilização dos recursos disponíveis em ambiente de programação LabVIEW, muitos destes exclusivos a esta linguagem, é necessário se ter o conhecimento de como proceder em suas implementações. A NI recomenda, principalmente, através de duas principais publicações, (NATIONAL INSTRUMENTS, 2009) e (NATIONAL INSTRUMENTS, 2014), em conjunto com o conteúdo apresentado em seus tópicos de ajuda presentes em seu software (NATIONAL INSTRUMENTS, 2012), que algumas práticas de programação sejam seguidas. Serão comentadas as principais utilizadas para este trabalho assim como exemplos sugeridos para um entendimento geral dos conceitos utilizados. Divididos em três partes, os tópicos tratados serão divididos em estrutura de armazenamento de dados, controles de processamento e estrutura de programa. Facilitando assim a combinação mais adequada da escolha dos elementos a serem utilizados em programação LabVIEW com um RTOS em conjunto com um FPGA.

2.5.1 Estruturas de armazenamento de dados

2.5.1.1 Constantes

Constantes numéricas são geralmente utilizadas para inializar um processo, um *shift register* por exemplo, ou quando um processo se torna conhecido e não necessita mais de modificações de maneira programática. Constantes são comumente utilizadas para configurações de VI's e sub VI's no LabVIEW. Podem ser de diversos tipos, qual for o interesse da classe a ser utilizada, como dados numéricos, *arrays*, etc. Na Figuras 5 são mostrados alguns tipos de constantes que serão utilizadas neste trabalho.

Figura 5 – Exemplos de constantes



Fonte: Produzido pelo autor

No item **a.** é mostrada uma constante do tipo I16, em **b.** uma constante do tipo *complex double*, no item **c.** uma constante do tipo *double* e no item **d.** um *array* constante com elementos do tipo *complex double*.

2.5.1.2 Controles

Controles são utilizados tanto para servirem de dados de entrada para um programa em execução, VI, ou para uma sub VI. Podem ser de qualquer classe. Podem ter sua configuração para uma faixa de valores possíveis a serem inseridos, desde que sejam configurados adequadamente. Sua modificação, quando apenas presente em uma VI principal, só pode ser modificada pelo usuário que utiliza a interface de usuário. Se o controle se encontrar fora de um *loop while*, *for* ou SCTL, não será utilizado seu valor atual, sendo que somente o registrado antes do início do processo será usado para a rotina escrita.

2.5.1.3 Indicadores

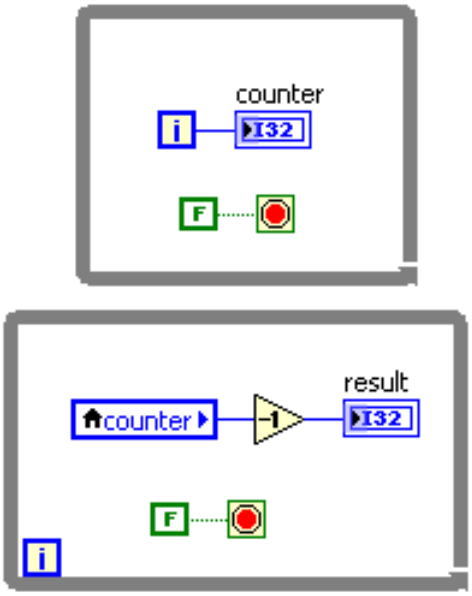
Indicadores exibem o resultado de determinados processo ocorridos com os dados de entrada, ou ainda em qualquer ponto do fluxo de dados. Da mesma maneira, podem ser utilizados como resultados de sub VI's que serão utilizados pela VI principal. Quando criados no *target*, FPGA, podem ser lido no *host*, RT, para monitoramento dos processos ocorridos na VI executada no FPGA.

2.5.1.4 Variáveis

Variáveis locais servem para ser escritas ou lidas em diferentes partes do *block diagram*. Quando a variável é um conjunto muito grande de dados, *arrays*, recomenda-se o uso de FIFO's. Utiliza-se quando não é possível fazer a ligação através de um "fio".

Recomenda-se seu uso para troca de dados entre *loops*. Pode-se ter diferentes domínios de *clock* ou não. Como a sua implementação é baseada em *flip-flops*, dos N valores escritos, N-1 valores serão perdidos, estando-se somente o último disponível para a leitura. Variáveis locais se tornam uma prática adotada quando não se deseja ter todos os valores escritos na variável, como por exemplo um loop que executa a leitura de uma variável modificada em outro *loop* de menor frequência.

Figura 6 – Leitura de uma váriavel em um *loop* paralelo através da variável local



Fonte: Produzido pelo autor

2.5.1.5 *Shift-registers*

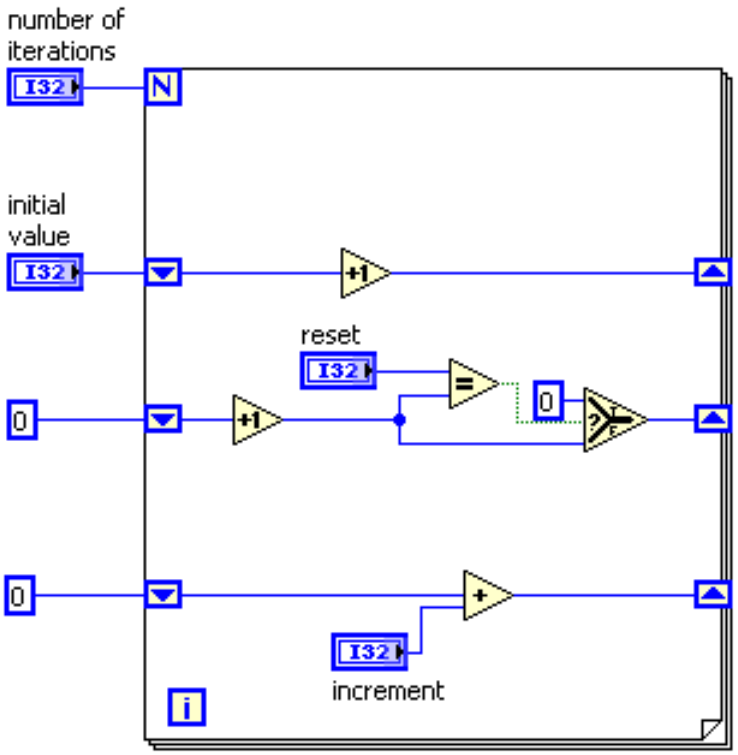
Shift Registers tem a função de armazenar valores entre iterações em um *loop*, tanto um *loop for* quanto um *loop while*. Aceitam várias tipos de dados, sendo os mais comuns numéricos.

Neste ambiente os contadores podem ser implementados de diversas maneiras. A primeira delas é utilizando a função incremento em conjuntos com o *shift register*, tanto de um *loop while* quanto de um *loop for*. Geralmente se utilizam contadores para controlar alguma outra ação que acontece dentro de uma VI ou sub VI, para isso, na maioria das ocasiões, eles possuem um valor final a que deseja se obter, ao atingir esse valor o contador é então reinicializado.

A Figura 7 exemplifica três contadores criados no *loop for*. O primeiro inializa-se em um valor qualquer e tem seu valor incrementado por uma unidade para cada iteração. Neste

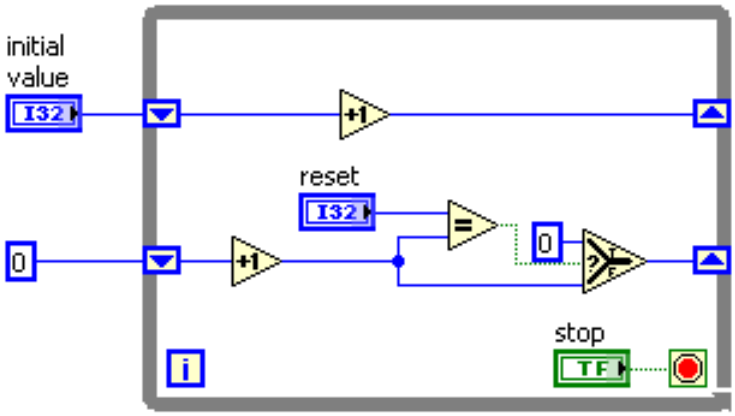
primeiro caso não há um valor final, o contador incrementará até o número de iterações ser satisfeito. No segundo contador a inicialização é feita em zero, há uma condição de *reset*, ou seja, o contador deve atingir um determinado valor e então ser resetado. Utiliza-se a lógica mostrada na figura, com as funções *Equal?* e *Select*. O terceiro contador é utilizado com incremento diferente de 1, podendo ser um valor qualquer. O mesmo também pode ser feito com o loop while, Figura 8.

Figura 7 – Contadores em um *loop for*



Fonte: Produzido pelo autor

Figura 8 – Contadores em um *loop while*

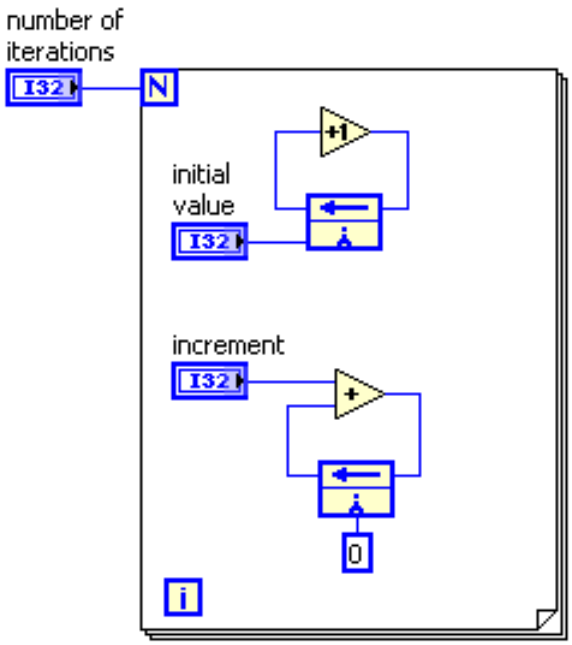


Fonte: Produzido pelo autor

2.5.1.6 *Feedback Node*

Assim como os *shift registers*, *feedback nodes* também servem para armazenar dados entre iterações, no entanto, não necessitam que "fios" sejam utilizados entre os extremos de um *loop*, o que pode trazer benefícios de entendimento para o código a ser executado, devido a melhor visualização. Também há a possibilidade de armazenar mais de um dado entre iterações, no entanto, o comportamento segue sendo o mesmo, um dado novo é inserido e um dado antigo é retirado, lido. Para operações envolvendo-se latência, torna-se útil utilizar esta funcionalidade, uma vez que melhoram a visualização do código, deixando-o mais enxuto. Podem ter duas direções de fluxo de dados, direita e esquerda para qual for o sentido conveniente do fluxo de dados. A Figura 9 exemplifica alguns dos mesmos contadores implementados com *shift registers*, Figura 7, com os *feedback nodes*.

Figura 9 – Contadores em um *loop for* com *feedback nodes*



Fonte: Produzido pelo autor

2.5.1.7 *Enum*

Para criar uma lista com *strings* associadas a números, a função *enum* é a mais utilizada. Recomenda-se seu uso em nomeação de estados de uma máquina de estados, por exemplo.

2.5.1.8 *Type Definitions*

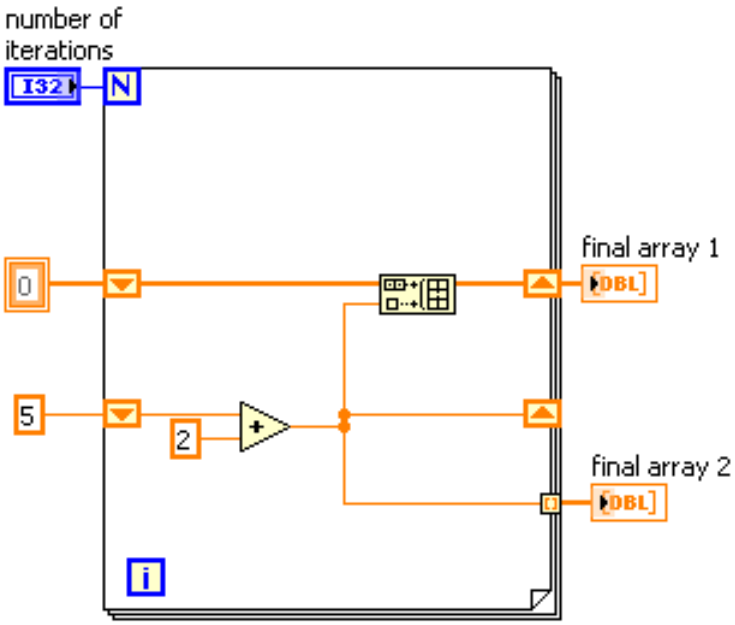
Type Definitions permitem que sejam identificados os tipos de dados corretos para um controle personalizado. A grande utilização desta funcionalidade é que uma vez alterando-se a configuração de um controle, todas as instâncias que contém este serão alteradas automaticamente. Torna-s muito útil, por exemplo, no desenvolvimento de fluxo de dados entre máquinas de estado em desenvolvimento, onde sua construção é feita por diferentes inserções de funcionalidades. Onde a cada nova inserção de dados os controles antigos são atualizados automaticamente, não havendo a necessidade de alterá-los um a um.

2.5.1.9 *Arrays*

Quando se deseja escrever ou registrar grandes porções de dados do tipo *array* na execução de uma VI, é necessário ocupar somente a memória necessária para isto, sem que

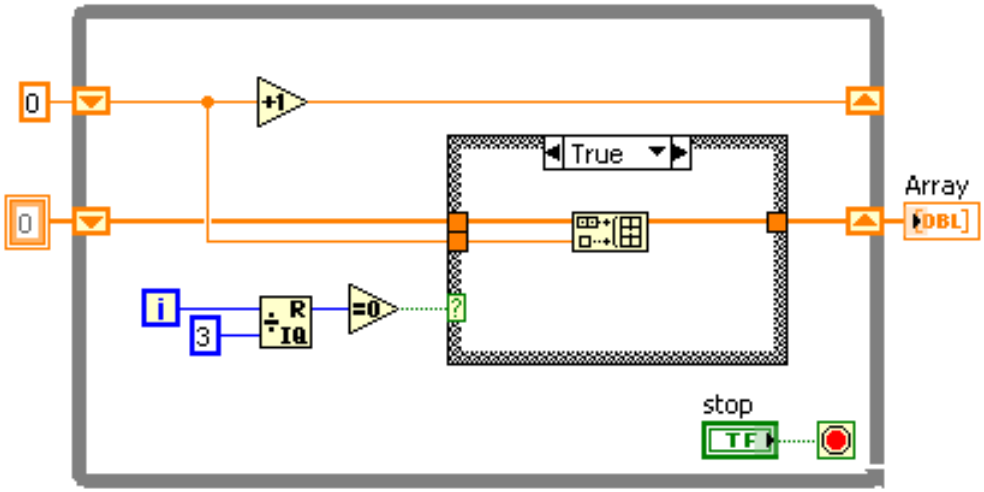
a execução do resto do código seja prejudicada ou que ocorra delays entre operações. A prancheta de operações de *arrays* fornece um grande número de funcionalidades ,no entanto, algumas de suas combinações entre si não são recomendadas. Quando apenas é necessário que um conjunto de dados seja gerado dentro de um *loop* e passado para a próxima iteração de laço, existem duas principais alternativas. A primeira é de se utilizar um *shift register* iniciado com valor constante e a estrutura *Build Array* no modo *concatenate inputs*, Figura 10, vide "final array 1". A segunda é de se utilizar o modo *ndexing*, tanto do *for* quanto do *while loop*. Ligando-se diretamente o "fio" aos extremos do *loop*, Figura 10, vide "final array 2". A segunda é a que exige menor recursos de memória, no entanto, para códigos em que o dado coletado esteja dentro de um *case structure* ela exigirá um valor *default* para o caso em que não for um dado desejado. Como exemplo, Figura 11, mostra uma *case structure* que coleta as apenas os valores de um contador nas iterações múltiplas de 3. A primeira opção é a mais econômica, uma vez que não a utilizando é necessário fazer alguma operação de seleção dentre os dados coletados. Caso não haja um *case structure* então o modo *indexing* é o recomendado a ser utilizado.

Figura 10 – *Arrays* sendo formados através da indexação



Fonte: Produzido pelo autor

Figura 11 – *Arrays* sendo formados através da concatenação



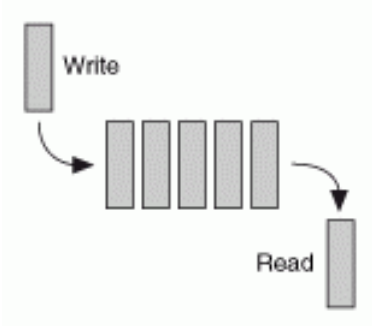
Fonte: Produzido pelo autor

Na operação de transferência de dados entre dois laços recomenda-se, a utilização dos blocos *Initialize Array* em conjunto com *Replace Array Subset* e um contador (NATIONAL INSTRUMENTS, 2012). Assim, a alocação de memória para cada iteração torna-se a mesma, o que diminui o número de recursos utilizados. Na inicialização do *array* há dois terminais que devem ser considerados, o tamanho do *array* e o tipo de dado que deve ser inicializado. Este preencherá todos os dados do *array*. Com o auxílio de um *Array Subset* substituem-se blocos de dados a cada iteração. O contador auxilia na contagem destes blocos, podendo-se dividir e escolher qual bloco de dados será escrito a cada iteração. Para uma VI sendo executada no RT e recebendo dados do FPGA através de FIFO's DMA, seu uso deve ser pareado com as RT FIFO's para otimizar a transferência de dados, diminuindo-se a *jitter*. Comenta-se mais sobre essa implementação no item RT FIFO's.

2.5.1.10 FIFO's

Para transferência de dados entre diferentes porções do FPGA, entre dispositivos ou diferentes partes de uma VI, uma solução recomendada é a utilização de *buffers*. Uma FIFO (*First In First Out*) é um método de organizar estes dados de maneira que o primeiro dado recebido, o mais antigo, será o primeiro a ser processado. Uma maneira de se visualizar este processo é exemplificado com a Figura 12.

Figura 12 – Empilhamento de uma FIFO



Fonte: (NATIONAL INSTRUMENTS, 2012)

Desta maneira há um controle sequencial do preenchimento dos dados. No LabVIEW há três diferentes categorias de FIFO's: *target-scope*, *target to host* e *host to target*. O primeiro realiza a transferência de dados entre diferentes localizações do FPGA, podendo pertencer a um mesmo domínio de *clock*. O segundo realiza a transferência de dados do FPGA para o sistema o RT. Finalmente, o terceiro envia dados do *host* (RT) para o FPGA. Seja qual for a configuração utilizada deve-se conhecer a sua implementação para se fazer uma escolha mais eficiente para a aplicação escolhida.

Há três tipos de implementação: *Flip-Flops*, LUT's e Block Memory. *Flip-Flops* são recursos disponíveis no FPGA para armazenamento de dados e também para funções lógicas e aritméticas, logo, são os recursos que mais serão utilizados para implementar a lógica de programação contida no BD e, portanto, podem não ser os mais disponíveis para armazenamento de dados. No entanto, são a implementação que garante a maior velocidade. Se utilizados para armazenamento, utilizam um grande espaço para tal, por isso é recomendado que sejam utilizados quando a quantidade de dados não for superior a 100 (NATIONAL INSTRUMENTS, 2014). LUT's acabam sendo mais eficientes em termos de armazenamento do que *Flip-Flops*, isto é, não ocupam tantos recursos para armazenamento, assim, para dados entre 100 e 300 bytes podem ser utilizadas. Tanto *Flip-Flops* quanto LUT's não podem ser utilizadas quando há transferência de dados entre dois domínios de *clock* diferentes. Para grandes quantidades de dados e múltiplos domínios de *clock* utiliza-se a *Block Memory*. São recursos específicos dentro do FPGA para armazenamento de dados. É a única implementação que é utilizada para fazer transferência de dados entre RT e FPGA.

Para se utilizar uma FIFO é necessário que esteja configurada corretamente.

Figura 13 – Bloco que configura a escrita de uma FIFO

Fonte: (NATIONAL INSTRUMENTS, 2012)

Figura 14 – Bloco que configura a leitura de uma FIFO

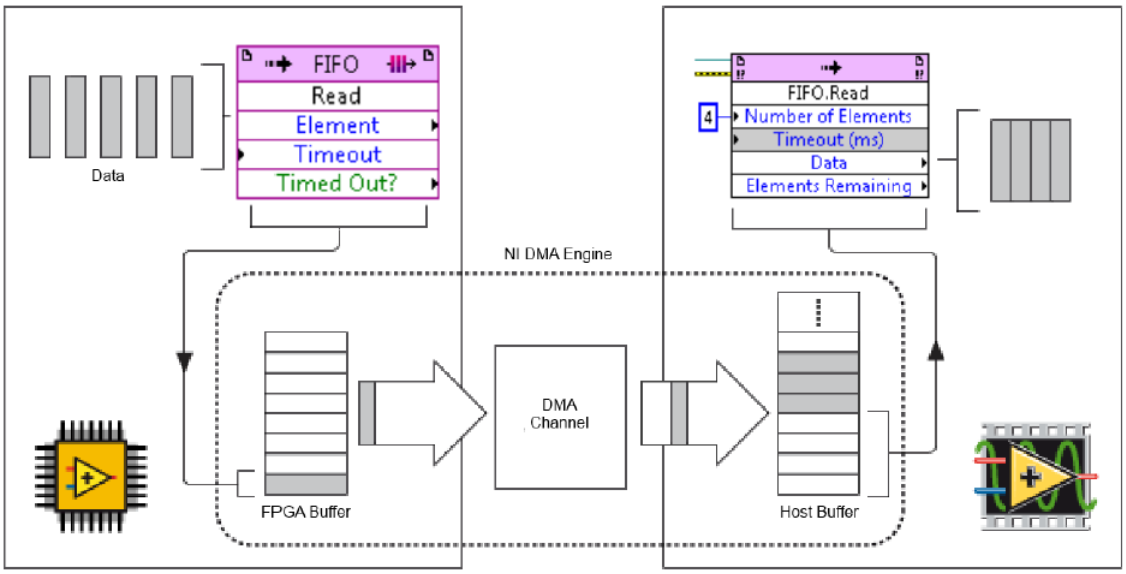
Fonte: (NATIONAL INSTRUMENTS, 2012)

Os terminais *FIFO In* e *FIFO Out* referem-se a qual FIFO está sendo utilizada e devem ser utilizados somente se mais algum item será configurado dentro do BD ligado a mesma. Na escrita dos dados é necessário que se coloque o dado que deseja ser armazenado na FIFO caso seja selecionada a funcionalidade *write*. Toda FIFO que for utilizada como *target-scope* deve ser escrita e posteriormente lida, não podendo deixar de possuir estas duas funções. O terminal de *Timeout* é utilizado para o tempo, em número de *ticks*, que, tanto a escrita quanto a leitura, devem esperar até haver espaço ou poder ser lido um dado. O valor de '-1' impede que a função entre em *timeout*; o valor 0 indica nenhuma espera. Se esta função estiver sendo utilizada dentro de um SCTL, é obrigatório o uso da constante 0.

O terminal *Timed Out?* escreve um valor verdadeiro se no caso da escrita não existir espaço para o armazenamento de um dado. No caso da leitura, *Timed Out?* é verdadeiro, caso não haja um dado novo para leitura, ou seja, FIFO vazia. Para algumas aplicações é necessário que se considere projetar uma lógica de *timeout* para as FIFO's para não haver perda de dados, *overflow* e *underflows* de *buffers* e também para que recursos lógicos não sejam eficientemente usados.

Um canal DMA (*Direct Memory Access*) funciona com dois *buffers*, um no *target* e outro no *host*.

Figura 15 – Estrutura de um canal DMA

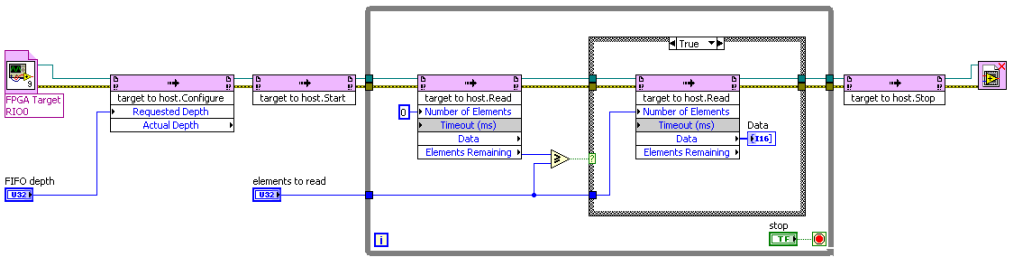


Fonte: (NATIONAL INSTRUMENTS, 2014)

No FPGA os elementos são armazenados um a um. No RT estes mesmo elementos são lidos em *arrays* de dados. Algumas diretrizes devem ser seguidas para a configuração para o tamanho destes *buffers* (NATIONAL INSTRUMENTS, 2014). A primeira é que, por definição, seu tamanho deve ser maior do que 10000 elementos e duas vezes maior do que o tamanho do *buffer* no FPGA. Caso encontrem-se problemas de *overflow* e *underflow*, utilize-se múltiplos de 4096 elementos para seu tamanho. Outra regra geral é que o tamanho do *buffer* no *host* deva ser pelo menos 5 vezes maior do que o do FPGA. Na maioria dos casos o tamanho do *buffer* no FPGA é de 1023 elementos, e assim deve permanecer a menos que encontrem-se fortes evidências de que é necessário alterar o seu valor. Isto se deve ao fato de que o FPGA possui recursos limitados e também porque pode ser executado múltiplas vezes mais rápido do que o *host*, logo, não devendo ser a primeira opção para diminuição de recursos a menos que estritamente necessário.

Para leitura de dados no *buffer* do *host* é recomendada a arquitetura expressa na Figura 16.

Figura 16 – Diagrama de blocos para leitura de dados no *host*



Fonte: Produzido pelo autor

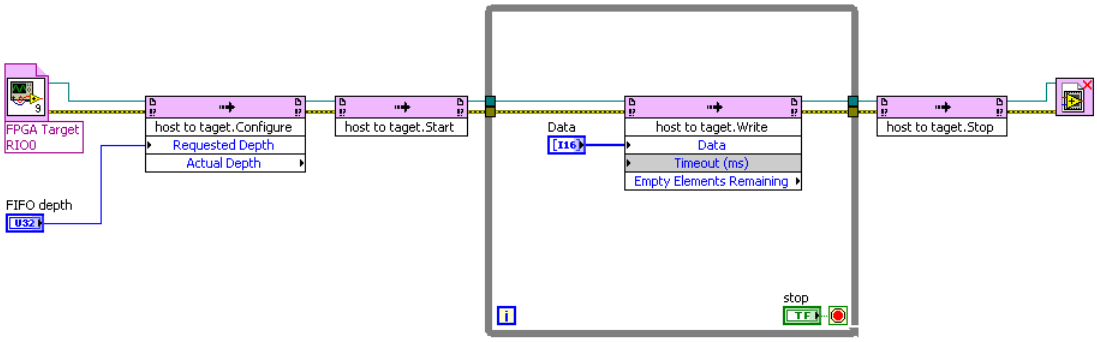
Primeiramente, é necessário que se utilize a função *Open FPGA VI reference* para escolher a VI que esteja no *target*. Em seguida, é necessário configurar a profundidade (tamanho) do *buffer* do *host* através da função *Invoke Method*, selecionado a FIFO de interesse no modo *Configure*, através do controle numérico *FIFO depth*. Para que o *buffer* do FPGA comece a escrever os valores no canal DMA é necessário utilizar outro *Invoke Method* para o modo *Start* para que os valores estejam disponíveis para leitura no *host*, pois somente após ter sido iniciada a FIFO, ela começará a fornecer dados ao *host*. Ao iniciar o *loop while*, usa-se o *Invoke Method* para configurar a FIFO no modo Leitura (*Read*). Em seu terminal *number of elements* coloca-se o valor de 0 elementos. Seu terminal de saída *Elements Remaining* conecta-se a um dos terminais de entrada bloco de *Greater or Equal?*. Em seu outro terminal, conecta-se um controle numérico. O resultado do bloco *Greater or Equal?* é conectado a uma *Case Structure* e em seu caso verdadeiro coloca-se novamente um *Invoke Method* no modo *Read*. Em seu terminal *number of elements* coloca-se o mesmo "fio"do controle numérico "elements to read"; ao terminal *Data* conecta-se um indicar dos dados de saída, *array*. Fora do *while loop* finaliza-se a FIFO, *Invoke Method* modo *Stop* e fecha-se a referência da VI que encontra-se no FPGA através da função *Close FPGA VI Reference*.

A ideia por trás desta estratégia é a de fazer a leitura da FIFO, ou seja, seu esvaziamento, através de um número constante de elementos. A *Case Structure* é utilizada para que somente quando o número de elementos for superior ou igual ao estipulado, *elements to read*, sua leitura seja feita. Caso contrário, a FIFO não é esvaziada. Desta maneira, previne-se que o número de elementos seja sempre distinto a cada leitura e também de que tenha-se um controle para posteriores operações com os dados de interesse desta FIFO.

Semelhante ao modo anterior, o lançamento de dados do *host* para o FPGA

acontece da mesma maneira, só que no sentido oposto. A Figura 17 exemplifica o modelo de programação a ser seguido.

Figura 17 – Diagrama de blocos para envio de dados ao *target*



Fonte: Produzido pelo autor

Mantém-se a mesma estrutura externa ao *loop while*, sendo a única diferença interna a ele. Para a transferência dos dados para o FPGA é conectado um controle de *array* ao *Invoke Method* no modo *Write*. O terminal de entrada de *Timeout (ms)* pode ser conectado a um valor numérico de acordo com as necessidades do projeto, assim como o terminal de saída *Empty Elements Remaining* pode ser utilizado ou não. Este apresente o número de posições vagas no *buffer* do *host* que será enviado ao *buffer* do *target*.

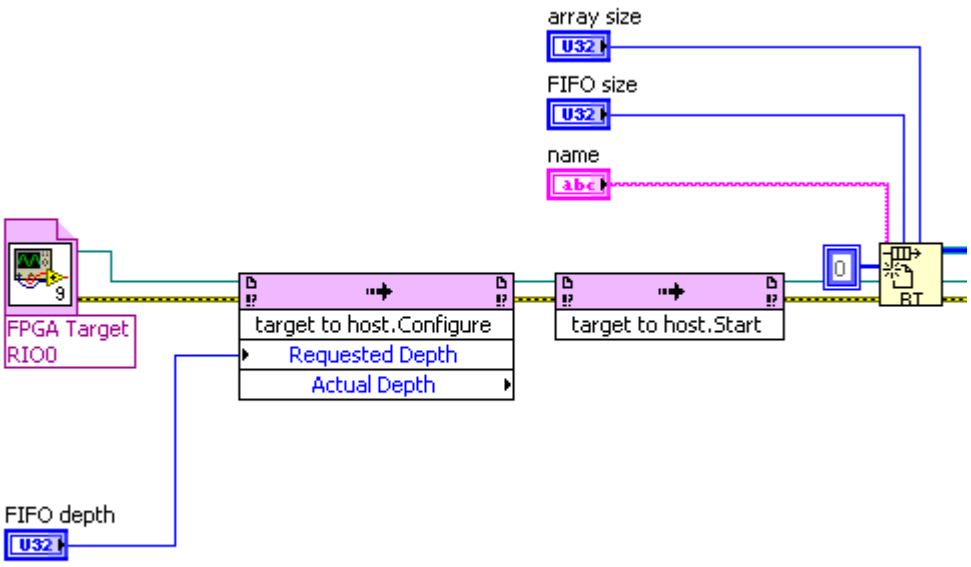
2.5.1.11 RT FIFO

Assim como as FIFO's disponíveis no FPGA eram utilizadas para transferir dados entre diferentes instâncias, as RT FIFO's são utilizadas unicamente dentro do RT. É uma função que propicia uma transferência de dados determinística entre VI's neste ambiente. É uma forma de comunicação com perdas, *lossy*, que reescreve os valores mais antigos armazenados por novos valores quando a FIFO encontra-se cheia. É comumente utilizada para transferir os valores adquiridos pelas FIFO's entre FPGA e *host* para um *loop* subsequente para processamento destes dados.

Para que isto seja feito o seguinte procedimento é adotado: cria-se um *array* constante de dimensão "RT FIFO size", juntamente com a função *Replace Array Subset* insere-se neste *array* os blocos de dados lidos pela FIFO no *host*. Um contador é utilizado para controlar o número de vezes que estes blocos de dados são inseridos no *array*. Assim, há uma relação entre o número de elementos lidos pela FIFO no host, *elements to read*, o índice final do contador, "x"e o tamanho da RT FIFO, que é "x*elements to read = RT FIFO size". Ou seja, controla-se quantas vezes a quantidade de dados lidos pode ser

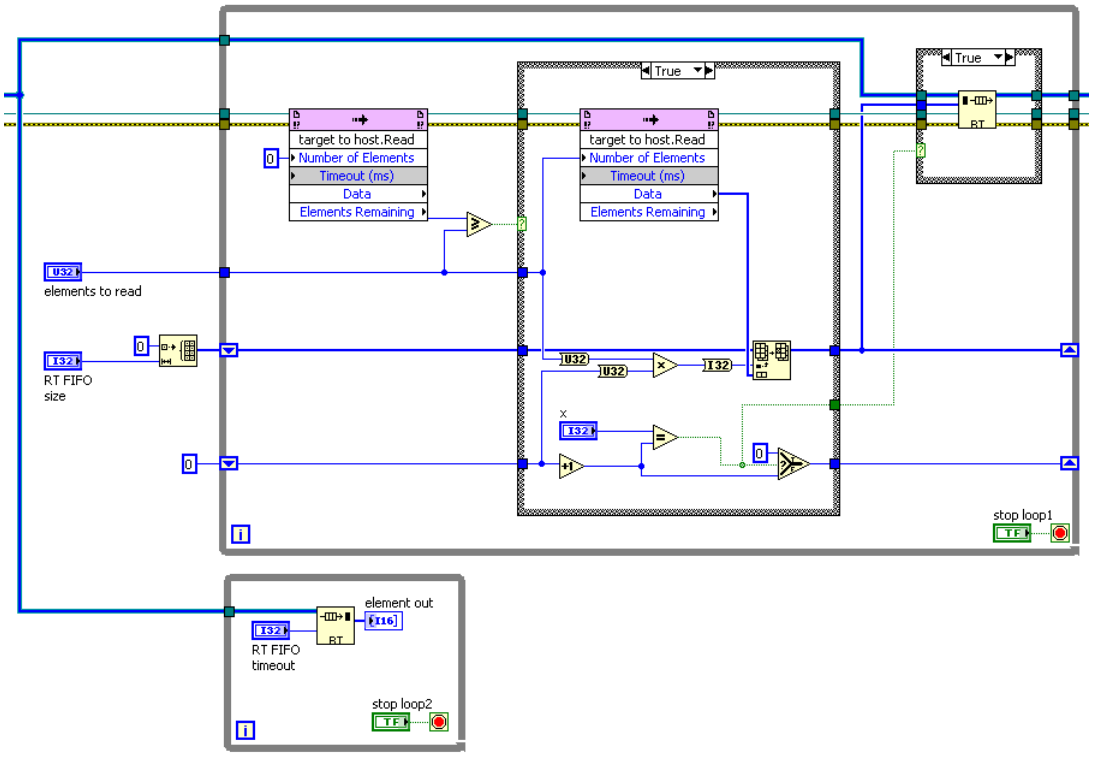
modificado neste *array*, quando este possuir apenas novos elementos o contador é zerado, dando um sinal positivo através da função *Equal?*. Este sinal então é passado à *case structure* onde encontra-se o bloco *RT FIFO write*, fazendo o armazenamento dos novos dados que serão lidos no *loop* mais inferior, "element out".

Figura 18 – Diagrama de blocos para configuração da FIFO de recebimento de dados do *target* e para a criação da RT FIFO



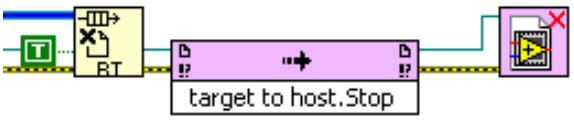
Fonte: Produzido pelo autor

Figura 19 – Diagrama de blocos para leitura da FIFO do *target* e transferência dos valores à FIFO RT



Fonte: Produzido pelo autor

Figura 20 – Encerramento da FIFO e RT FIFO



Fonte: Produzido pelo autor

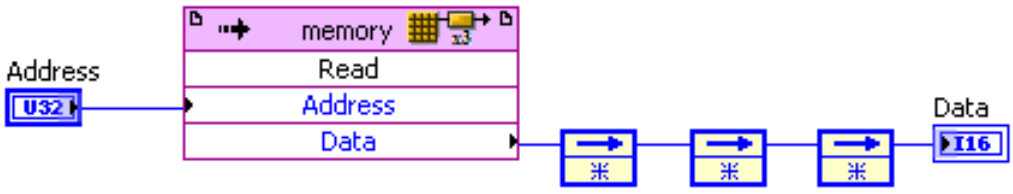
O modo do empilhamento dos elementos na RT FIFO ocorre conforme o indicado na Figura 19. Antes de iniciar o *loop* em que a FIFO realiza a transferência de dados, incializa-se um *array* com um número de elementos definidos pelo controle numérico "RT FIFO size", prática explicada no item 2.5.1.9. Este *array* será utilizado para a transferência dos valores recebidos pela FIFO para a RT FIFO. Com o auxílio do contador incializado em zero, tópico tratado no item 2.5.1.2, tem-se o seu valor final inserido pelo controle "x";

sua inicialização é feita novamente com o número zero. Utiliza-se uma função *Multiply* entre o número de elementos lidos pela FIFO no *host*, *Elements To Read* e pelo contador.

2.5.1.12 *Memory Item*

Memory Items podem armazenar dados em endereços fixos e específicos. Para a aplicação deste trabalho, utilizou-se a implementação *Block Memory*, esta permite a utilização de leitura em diferentes domínios de *clock*. Também são um recurso próprio do FPGA para armazenamento de dados, e por isso não consomem nenhuma lógica adicional. Tem a vantagem, sobre as demais (*Flip-Flops* ou LUT's) de compilar em *clocks* mais altos (NATIONAL INSTRUMENTS, 2012). Sugere-se que se utilize-a preferencialmente entre as demais a menos que se deseje ter outras vantagens das outras. Neste trabalho será utilizada apenas a função de leitura da memória. Para garantir que sua implementação seja possível em um *clock* elevado é necessário levar em consideração a sua latência. Assim, a memória necessita de alguns ciclos de *clock* para colocar um valor válido em seu terminal de saída, para que nenhum valor seja perdido, é necessário colocar ligado a sua saída *feedback nodes* não inicializados (NATIONAL INSTRUMENTS, 2012). Sua quantidade deve ser igual ou maior do que o número de ciclos de latência que aparecem no campo da figura. Logo, para este item, sua disposição no BD seria como a da Figura 21.

Figura 21 – *Feedback Nodes* não inicializados conectados ao terminal de saída do memória



Fonte: Produzido pelo autor

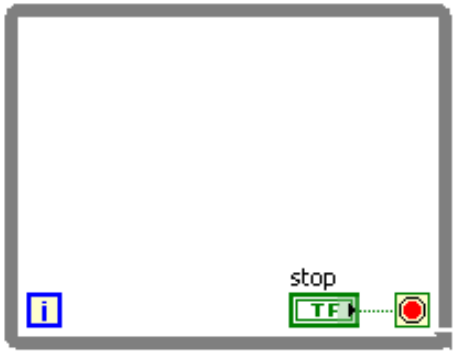
2.5.2 Controles de processamento

2.5.2.1 *while loop*

Um *loop while* executa enquanto uma condição de finalização não for atingida. Dentro de sua estrutura há um terminal de finalização chamado *loop condition* que deve ser sempre conectado a alguma condição que deve ser satisfeita. Este *loop* executa pelo

menos uma vez. Assim como no *loop for*, o terminal "i" também está presente, e seus limites vão desde 0 até o número de vezes que o *loop* executa menos um.

Figura 22 – *While loop*

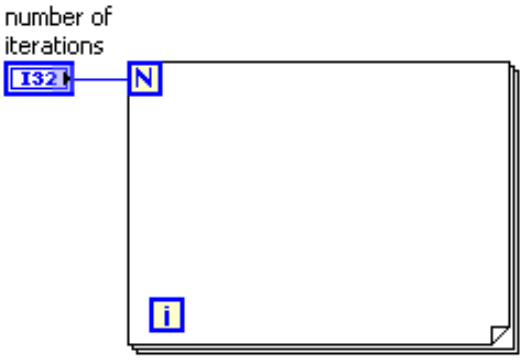


Fonte: Produzido pelo autor

2.5.2.2 *For loop*

Um "*loop for* antes de ser acionado precisa receber o valor de quantas iterações terá. Através de seu terminal "N" insere-se a quantidade de vezes que deve ser necessário para que o *loop* se repita. Dentro dele há o contador intrínseco ao *loop*, "i", que pode ser utilizado ou não para alguma função dentro dele. A relação entre "i" e "N" é de N = i - 1. Na Figura 23 abaixo há um exemplo de um *loop* for que executará cinco vezes.

Figura 23 – *For loop*



Fonte: Produzido pelo autor

2.5.2.3 *Single-cycle Timed Loop* (SCTL)

Em um SCTL o tempo de execução corresponde ao *clock* do FPGA que é especificado. Ou seja, para um SCTL que é configurado para um clock de 40 MHz, cada iteração terá uma duração de 25 ns. Para toda porção de código que se encontra dentro do SCTL, tendo-se a compilação validada, é garantido que ela execute neste período em todas as suas iterações. A vantagem de se ter um SCTL ao invés de um *loop while* executando com um *Loop Timer Express* é que há total garantia de que cada iteração terá sempre o mesmo comportamento. Uma das desvantagens de se utiliza-lo é de se ter um número menor de funções disponíveis para seu uso, nem todas as que podem ser executadas dentro de um *loop while* podem ser utilizadas, como por exemplo entradas e saídas analógicas. Isto está relacionado com o fato de como o SCTL realiza as operações lógicas. Para um *loop while* *flip-flops* são utilizados para armazenar resultados intermediários. Em um SCTL isto não ocorre, pois a lógica é implementada diretamente em *hardware*, e todas as funções são implementadas de maneira a serem executadas em um ciclo de *clock*.

2.5.2.4 *Auto-Indexing*

Automaticamente ao se conectar um *array* de N elementos a um *loop for* o LabVIEW indexará os valores, isto quer dizer que a cada iteração será selecionado um dos elementos do *array* em ordem crescente, iniciando-se na posição 0, primeira, até a N-1, última. Logo, o número de iterações que este *loop* fará está relacionado com a dimensão do *array*, assim, seu o terminal "N" do *loop* não necessita ser conectado a nenhum valor a menos que se queira executa-lo um número de vezes menor do que a dimensão do *array*. Esta função também é possível de ser utilizada para construir-se um *array*, onde sua dimensão será igual ao número de iterações. Esta função de indexação ocorre automaticamente pois em um *loop for* naturalmente se espera realizar operações elemento a elemento para um *array*.

2.5.2.5 High Throughput Math Functions

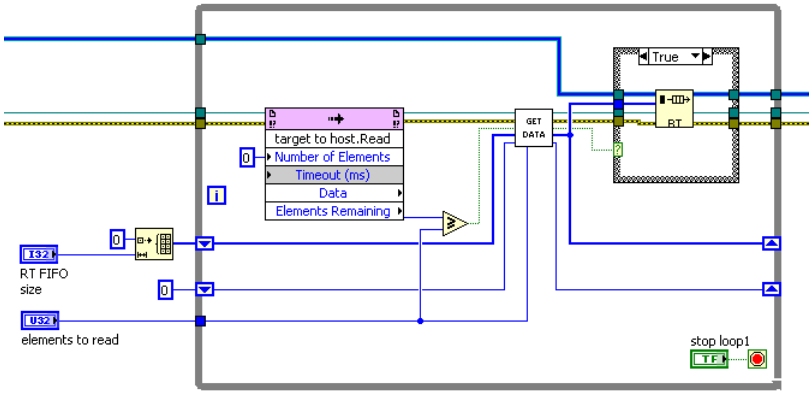
Esta paleta de funções é utilizada quando é necessário que altas taxas de dados sejam processadas. Para que se utilize a máxima funcionalidade e recursos do FPGA, estas possuem a possibilidade de realizar os cálculos através de *pipeline*. Assim, possuem terminais de *handshaking*, notificando-se quando é possível receber um dado para cálculo, quando a operação está finalizada, entre outras funções. Quando a operação não pode ser realizada em um único ciclo de *clock*, os terminais de *handshaking* devem ser utilizados No entanto, no escopo deste trabalho, eles não serão utilizados pois todas funcionalidades utilizadas desta paleta foram possíveis de serem implementadas em um único *tick*.

2.5.3 Estruturação do programa

2.5.3.1 Sub VI's

Sub VI's são porções de código que podem ser agrupadas em uma VI e colocadas dentro de outra VI. Recebem este nome porque quando encontram-se dentro da VI principal, acabam sendo subordinadas às principais. É indicado seu uso quando o código principal torna-se composto por uma grande quantidade de elementos, tornando a edição do código trabalhosa. Assim, pode-se escolher a porção mais conveniente e agrupá-la em outra VI com suas entradas e saídas estando disponíveis para o mesmo diagrama principal. Isto torna o código mais rápido de ser entendido. É recomendado que se nomeie este novo bloco com um nome adequado e que os terminais de entrada e saída estejam de acordo com o lugar onde estará esta VI ligada. É importante verificar se a escolha desta sub VI pode ser utilizada para uso posterior em outra aplicação ou até mesmo dentro da mesma VI.

Figura 24 – *For loop*A figura apresenta uma possibilidade de sub VI criada para o mesmo diagrama de blocos da Figura 19

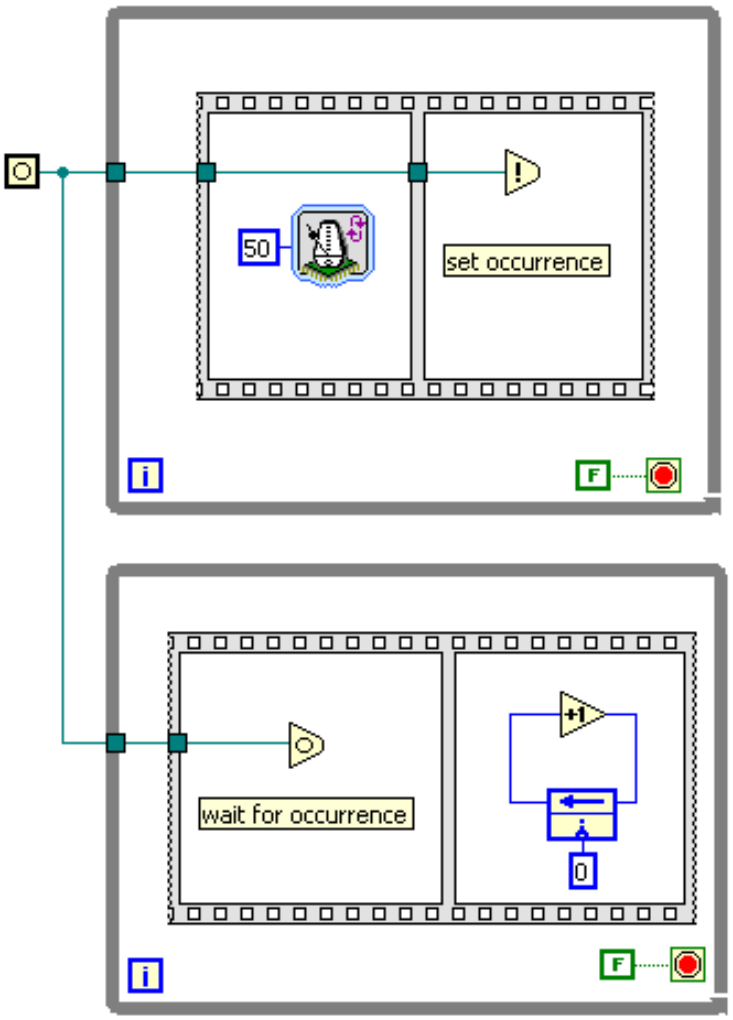


Fonte: Produzido pelo autor

2.5.3.2 Ocorrências - *Occurrences*

São utilizados para controlar e sincronizar atividades síncronas distintas. Relacionando dois *loops* paralelos através de uma occurrence evita-se que haja uma super utilização de recursos de processamento, pois estando um *loop* a espera de uma occurrence ele permanece em estado de espera, *idle*, até que a mesma seja recebida. Utilizan-se duas VI's deste bloco, *Set Occurrence* e *Wait For Occurrence*. A primeira coloca-se na localização apropriada do diagrama de blocos que deseja-se lançar uma sinalização, com a segunda espera-se por este sinal e após recebê-lo, realiza-se a atividade em espera.

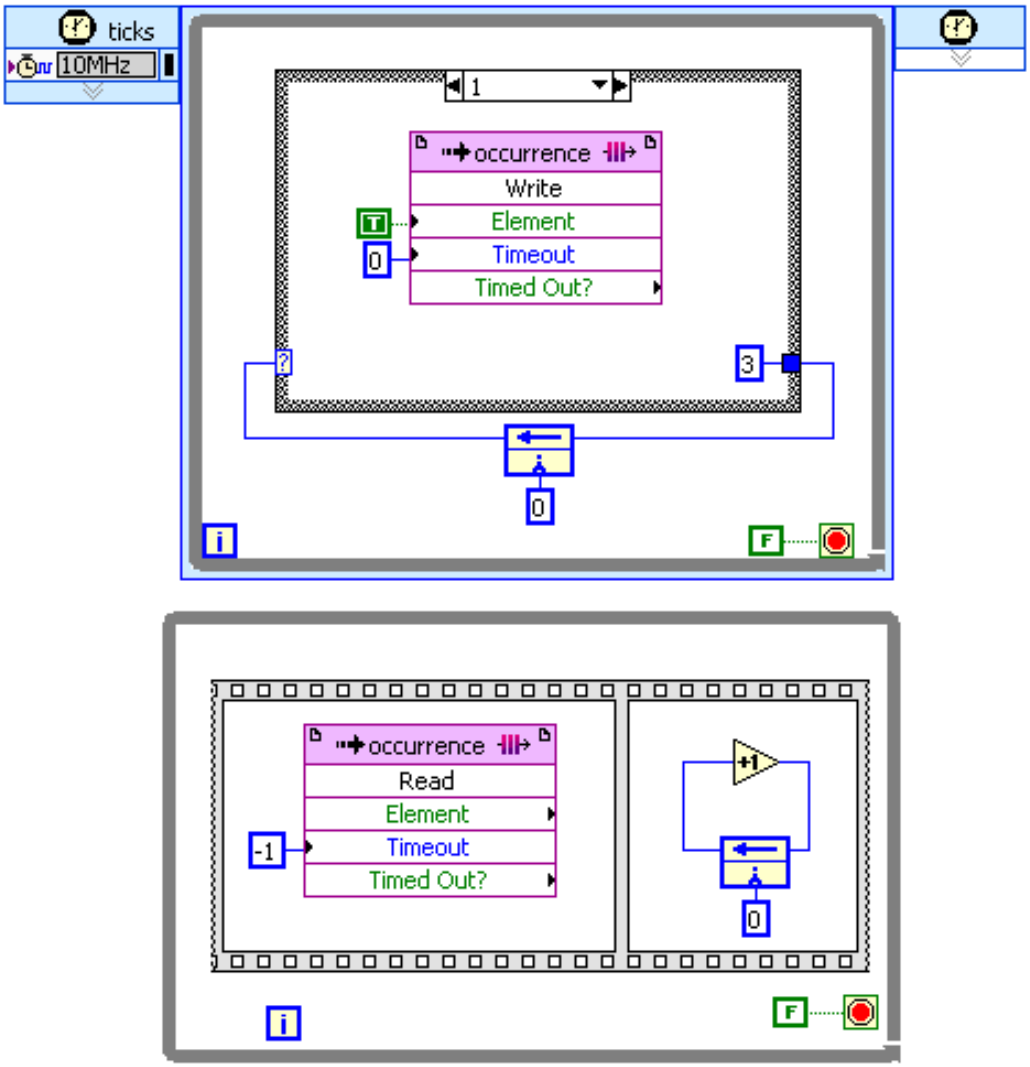
Figura 25 – Um *loop* operando pelo comando de uma ocorrência gerada em outro laço



Fonte: Produzido pelo autor

Na Figura 25 há um exemplo de um contador que está sincronizado com a execução de um *loop* paralelo. Seu contador só irá incrementar quando o comando *Set Occurrence* for acionado. No FPGA este comando só é válido para mesmos domínios de *clock*. Porém, existe uma maneira muito semelhante de se gerar uma ocorrência através da utilização de FIFO's na configuração de *target-scope* para domínios de *clocks* distintos. Através de um sinal de teste, com o terminal *Timeout* da FIFO configurada para -1, ou seja, espera infinita. A FIFO ficará em modo de espera até que haja o recebimento de um novo dado ainda não lido. Escolhe-se preferencialmente um sinal booleano para tal finalidade. A Figura 26 mostra o BD para que isto ocorra.

Figura 26 – Ocorrência através de um *dummy signal*

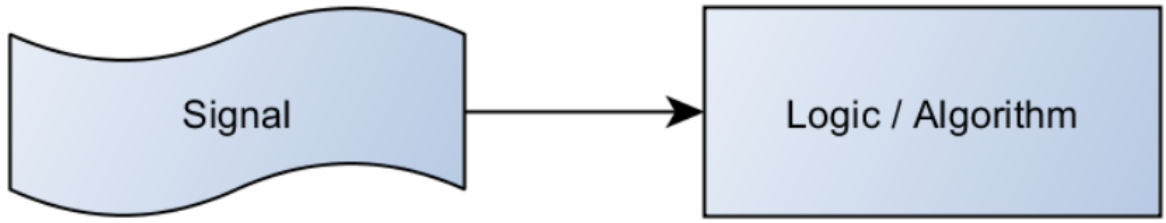


Fonte: Produzido pelo autor

2.6 *Latency*, *Throughput* e *Jitter*

Em aplicações que envolvam medidas ou monitoramento de algum sinal proveniente do mundo físico é necessário conhecer o tempo que cada tarefa correspondente do sistema de medição ocorre. Ou seja, quais são todas as etapas que existem para que uma única amostra do sinal seja levada ao *software* que fará o seu processamento, Figura 27.

Figura 27 – Sistemas que realizam medidas são projetados para que as amostras sejam levas para um algoritmo



Fonte: (HEIM, 2014)

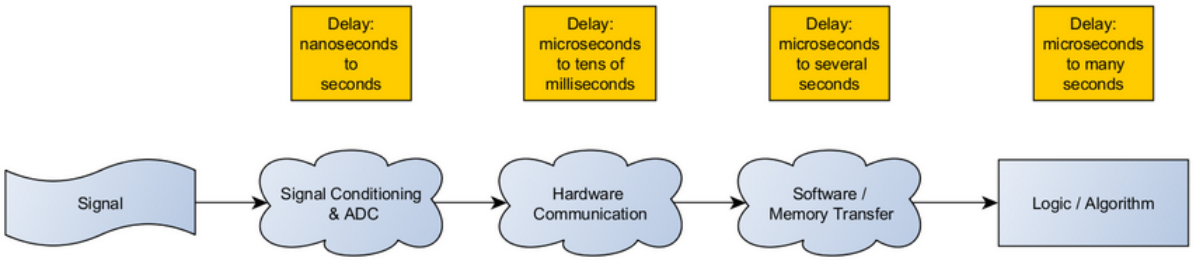
Qualquer que seja o sinal adquirido e tenha o objetivo de ser tratado digitalmente, alguns processos são comumente utilizados, Figura 28. Para a conversão do sinal analógico em sinal digital utiliza-se ADC's, e este causa o primeiro atraso. Após ter o sinal convertido na saída do ADC, existe a necessidade de levar este sinal ao *software*, isto é feito através da comunicação do *hardware*, trilhas físicas do circuito por onde passarão estes sinais elétricos, este, o segundo atraso. O sinal sendo entregue ao *software* que realizará os processamentos seguintes, haverá o terceiro atraso, que envolve o tempo necessário para que este dado seja colocado em algum tipo de memória para ser acessado quando necessário. Ainda há o quarto atraso, que envolve o tempo necessário que o *software* necessitada para ter um resultado válido do seu processamento sobre aquela única amostra inicial.

Figura 28 – Três classes de *delay*

Fonte: (HEIM, 2014)

Para cada um destes processos existe uma possível margem de tempos de atraso que podem ocorrer, estes são mostrados na Figura 29.

Figura 29 – Três classes de *delay* e seus respectivos tempos



Fonte: (HEIM, 2014)

Latência é a quantidade de tempo que um processo demora a acontecer, desde a sua entrada até a sua saída. Para a maioria das situações é considerado o tempo necessário para que um sinal entre no ADC e saia no DAC.

Throughput é a taxa com que o processo pode processar os dados de entrada. Está relacionada com a quantidade de amostras que podem ser recebidas pelo sistema, normalmente utiliza-se a unidade amostras por segundo, no inglês *samples per second* (S/s).

Para melhor o rendimento do processo podem ser utilizadas duas estratégias para aumentar o *throughput* através de *software timing* ou *hardware timing* (HEIM, 2014). Para esta aplicação será utilizada a segunda. Por se tratar de um FPGA o sistema possui um *clock* implementado em *hardware*, o que o torna preciso. Levando em conta esta precisão, amostras são colhidas e inseridas nas FIFO's. Através desta estratégia é possível tratar grandes conjuntos de dados, geralmente *arrays*, onde o tempo entre amostras se reduz comparativamente a amostras adquiridas uma a uma.

Já para a latência a estratégia para diminuí-la envolvem duas opções, um sistema operacional em tempo real ou um FPGA. *Jitter* é o termo que se dá quando há variação no tempo da latência. RTOS são especificamente desenvolvidos para diminuir esta variação. Um FPGA pode reduzir significamente o tempo de atrasos entre as entradas e saídas do sistema com os conversores e trilhas de circuito. Isto ocorre porque o *software* que fará o processamento com os sinais de entrada encontra-se diretamente configurado em *hardware*, pois conta com todas as funções do *software* implementadas nesta porção, excluindo assim a necessidade de comunicação das entradas e saídas com outra porção de hardware que fará o processamento por um *software* externo, geralmente o que ocorre quando o sistema não é implementado em FPGA.

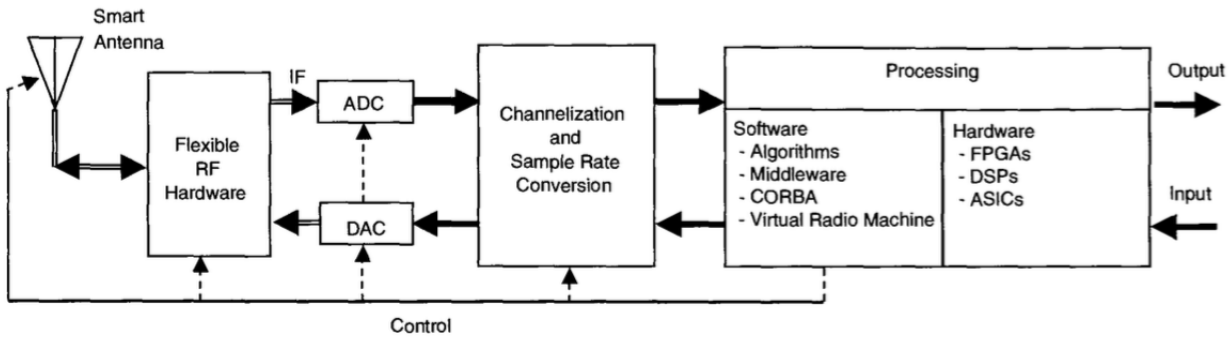
Nem sempre será possível conciliar a máxima eficiência tanto para latência e *throughput* ao mesmo tempo, pois geralmente diminuindo-se a latência, ou seja, tornando

o sistema mais veloz, pode-se diminuir a *throughput*. O mesmo ocorre quando aumenta-se a *throughput*, isto é, mais amostras podem ser processadas, aumenta-se a latência, há um aumento do tempo necessário para processar estas amostras. No entanto, tendo-se disponível estes dois recursos, é possível fazer escolhas de quando um processo será mais crítico que outro para avaliar a combinação dos dois recursos para que se possa atingir o objetivo da aplicação.

3 Uma arquitetura de SDR

Como descrito em , um SDR tem essencialmente as suas funções operativas implementadas em *software*. Uma generalização para a arquitetura de um SDR pode ser estruturada conforme a Figura 30.

Figura 30 – Diagrama de blocos genérico para a arquitetura de um SDR



Fonte: (REED, 2002)

Os blocos *Processing*, *Channelization and Sample Rate Conversion* e ADC/DAC compõe a parte do SDR voltada à programação. O bloco *Flexible RF Hardware* corresponde o chamado *front-end* do rádio. Esta parte que faz com que o sinal processado possa ser transmitido nas faixas de RF. Neste trabalho não será estudada e nem utilizada a parte que diz respeito à RF, assim, o estudo será concentrado no desenvolvimento do *software* e na implementação e escolha do *hardware*. *Output* e *Input* correspondem a possíveis entradas e saídas de dados que desejam ser enviados e recebidos. No caso deste trabalho, correspondem a sinais de áudio.

A implementação desenvolvida teve como modelo a apresentada por (KEHTARNAVAZ; MAHOTRA, 2010), onde utiliza-se a linguagem de programação LabVIEW implementada em FPGA. Onde buscou-se adaptar o trabalho feito para uma configuração mais distante de simulações, criando assim uma relação entre o mundo analógico e digital. No entanto, a base para a criação do *software* é muito semelhante a apresentada no livro, onde expõe uma sequência de operações que são realizadas sob o sinal para que possa ser amostrado, processado, transmitido e recuperado, configurando assim um sistema de comunicação. O sinal amostrado é um sinal analógico, posteriormente à amostragem ele configura-se um sinal digital, e assim, é as operações realizadas sobre ele são as de:

a. Modulação (Codificação);

b. *Upsampling*;

c. Modelagem de pulso;

d. Transmissão;

e. Recepção;

f. Filtragem;

g. Demodulação;

h. *Downsampling* e

i. Reconstrução (Decodificação).

De acordo com (KEHTARNAVAZ; MAHOTRA, 2010), os itens **a.**, **b.** e **c.** constituem o transmissor. Nestes, é feito o processo de modulação IQ (QAM), responsável por levar um sinal à banda passante através de dois sinais da banda base; o processo de *upsampling*, onde insere-se amostradas não pertencentes ao sinal (*zero-stuffing*) para aumentar a sua frequência de amostragem e por fim a modalgem de pulso, necessária para evitar a interferência intersimbólica. Os itens **d.** e **e.** configuram um canal aditivo de ruído branco gaussiano. **f.** é onde ocorre a recuperação da parte imaginária do sinal transmitido, processo que se faz necessário para a modulação de amplitude em quadratura, onde é realizada através da transformada de Hilbert. No item **g.** é feita a demodulação do sinal, processo que recupera os sinais da banda base a partir da banda passante, ou seja, onde são recuperados os sinais em fase e quadratura. Após, é feito o *downsampling* do sinal, isto é, descarta-se as amostras do sinal, essencialmente aquelas inseridas no *upsampling*. Assim, diminui-se a frequência de amostragem do sinal recebido, recuperando a original (amostrado). Tendo os dois sinais sido recuperados, parte para o último processo, o de reconstrução do sinal, a partir da decodificação do sinal, isto é, o mapeamento dos valores de IQ para os valores do sinal original.

Para cada um dos processos citados acima será feita uma explanação teórica de como acontecem e do porquê se fazem necessários. Buscando-se uma melhor compreensão dos assuntos abordados por estes processos, se faz necessário uma pequena introdução de alguns conceitos importantes para operações e manipulações de sinais, necessários para o entendimento da implementação que será exposta no Capítulo 4.1.

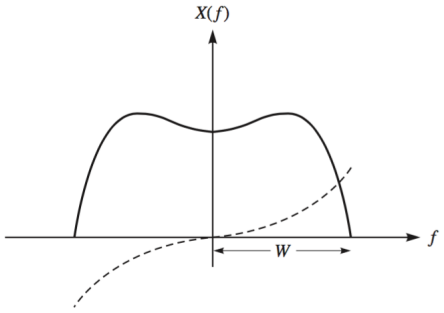
3.1 Conceitos introdutórios

Serão mencionados algumas definições e notações para o equacionamento presente nas seções seguintes. Buscando desta maneira uma relação mais imediata entre cada processo.

3.1.1 Sinal da banda base - *bandbase signal* ou *lowpasss signal*

Um sinal pertencente à banda base, $x(t)$, é aquele que tem seu espectro de frequências, $X(f)$, centrado em zero. Sua largura de banda, *bandwith* - W, é a faixa de frequências ocupadas pelo sinal.

Figura 31 – Espectro de frequência do sinal da banda base

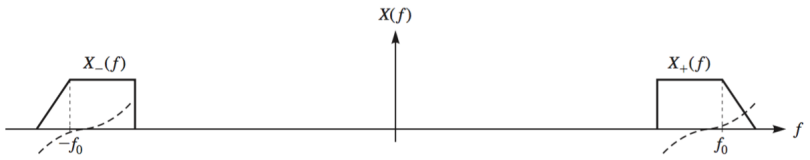


Fonte: (PROAKIS, 2008)

3.1.2 Sinal da banda passante - *bandpass signal*

Um sinal real pertencente à banda passante, diferentemente de um sinal da banda base, possui seu espectro de frequências centrado em uma frequência f_0 diferente de zero. Não é necessário que f_0 esteja exatamente no centro do espectro. É importante reassaltar que para recuperar o sinal original, não é necessário conhecer a parte negativa do espectro de frequências. O conhecimento da porção positiva do espectro é suficiente para a recuperação da informação do sinal.

Figura 32 – Espectro de frequência de um sinal real da banda passante. $X_-(f)$ e $X_+(f)$ representam os espectros de frequência negativo e positivo de $x(t)$

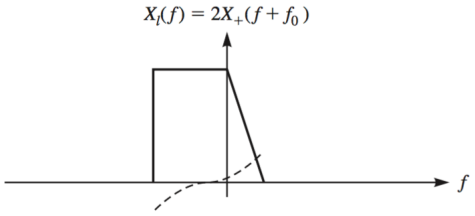


Fonte: (PROAKIS, 2008)

3.1.3 Sinal equivalente da banda base para sinais da banda passante

De acordo com (PROAKIS, 2008), dado um sinal $x(t)$ podemos definir sua representação analítica ou seu pré-envelope como $x_+(t)$, cuja transformada de Fourier é $X_+(f)$. Um sinal analítico é aquele que contém apenas frequências positivas, conforme Figura 33. Geralmente $x_+(t)$ é um sinal complexo.

Figura 33 – Espectro de frequência do sinal analítico $x(t)$



Fonte: (PROAKIS, 2008)

Desta maneira:

$$x_+(t) = \mathscr{F}^{-1}[X_+(f)] \tag{3.1}$$

reescrevendo apenas para frequências maiores que zero

$$x_+(t) = \mathscr{F}^{-1}[X(f)u_{-1}(f)] \tag{3.2}$$

aplicando a transformada inversa de Fourier

$$x_+(t) = x(t) \star \left(\frac{1}{2}\delta(t) + j\frac{1}{2\pi t}\right) \tag{3.3}$$

onde $\star$ representa a operação de convolução entre os dois sinais. Assim:

$$x_+(t) = \frac{1}{2}x(t) + \frac{j}{2}\hat{x}(t) \tag{3.4}$$

onde $\hat{x}(t)$ representa a transformada de Hilbert para $x(t)$, que será discutido em mais detalhes no item 3.5.

Define-se $x_l(t)$ como o sinal equivalente de banda base, ou sinal do envelope complexo de $x(t)$, Figura 33, aquele que possui um espectro de frequências dado por $2X_+(f + f_0)$.

$$2X_+(f + f_0)u_{-1}(f + f_0) \tag{3.5}$$

Aplicando o teorema da modulação da transformada de Fourier para o sinal $x_l(t)$, obtemos:

$$x_l(t) = \mathscr{F}^{-1}[X_l(f)] \tag{3.6}$$

resultando em

$$x_l(t) = 2x_+(t)e^{-j2\pi f_0 t} \tag{3.7}$$

e como $x_+(t)$ pode ser reescrito de acordo com a Equação 3.8

$$x_l(t) = (x(t) + j\hat{x}(t))te^{-j2\pi f_0 t} \tag{3.8}$$

expandindo a exponencial

$$x_l(t) = (x(t)\cos 2\pi f_0 t + \hat{x}(t)\sin 2\pi f_0) + j(\hat{x}(t)\cos 2\pi f_0 t - x(t)\sin 2\pi f_0) \tag{3.9}$$

As partes real e imaginária de x_l são chamadas, respectivamente, de componente em fase, *in-phase component*, e componente em quadratura, *quadrature component*, de $x(t)$. Ambas são sinais reais e recebem a representação $x_i(t)$ e $x_q(t)$. Desta maneira:

$$x_l(t) = x_i(t) + jx_q(t) \tag{3.10}$$

Comparando as equeações 3.9 e 3.10, temos

$$x_i(t) = x(t)\cos 2\pi f_0 t + \hat{x}(t)\sin 2\pi f_0 \tag{3.11}$$

$$x_q(t) = \hat{x}(t)\cos 2\pi f_0 t - x(t)\sin 2\pi f_0 \tag{3.12}$$

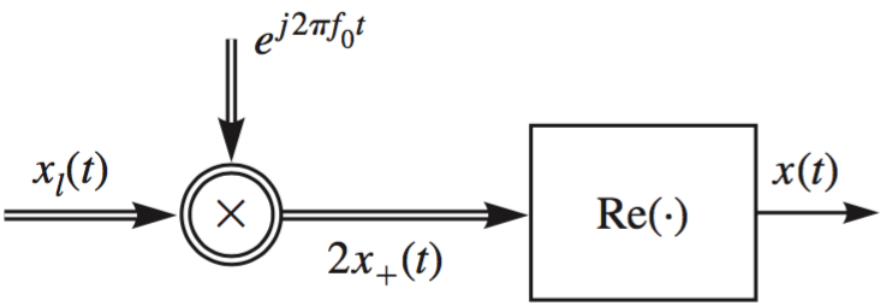
Das Equações 3.11 e 3.12, isola-se $x(t)$ e $\hat{x}(t)$

$$x(t) = x_i(t)\cos 2\pi f_0 t - x_q(t)\sin 2\pi f_0 \tag{3.13}$$

$$\hat{x}(t) = x_q(t)\cos 2\pi f_0 t + x_i(t)\sin 2\pi f_0 \tag{3.14}$$

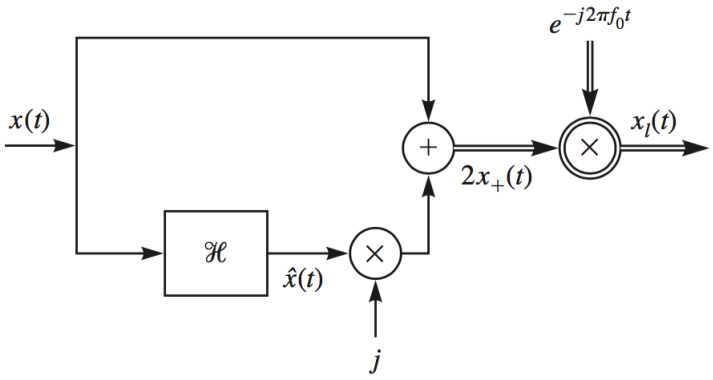
Assim, qualquer sinal da banda passante $x(t)$ pode ser expresso em termos de dois sinais da banda base. O processo de levar sinais da banda base para a banda passante é definido como modulação, representado na Figura 34. A relações dadas em 3.13 e 3.14, de representar sinais da banda passante em termos da banda base definem a modulação. O processo inverso, de recupear os sinais que compõe o envelope complexo, é definido como demodulação, conforme Equações 3.11 e 3.12, representado na Figura 35

Figura 34 – Esquema da implementação da modulação. Linhas duplas representam um sinal coplexo e linhas simples um sinal real



Fonte: (PROAKIS, 2008)

Figura 35 – Esquema da implementação da demodulação. Linhas duplas representam um sinal coplexo e linhas simples um sinal real



Fonte: (PROAKIS, 2008)

3.2 Interpolação - *Interpolation* e *Upsampling*

Upsampling é o processo de inserir amostras de valor zero entre amostras originais para aumentar a frequência de amostragem. Também pode recebe o nome de *zero-stuffing*. Já o fenômeno de interpolação, *interpolation*, caracteriza-se pelo processo de *upsampling* seguido de filtragem. Ela se torne necessária para remover imagens espectrais indesejadas.

Assim como no fenômeno da dizimação, uma das motivações para se fazer o *upsampling* é para adaptar a saída do sistema conectado a outro, só que neste caso operando a uma frequência mais alta.

Suponha o mesmo sinal citado em 3.6, mas agora deseja-se diminuir o período de amostragem, de maneira que $T_i = T/L$, assim, de acordo com (OPPENHEIM; SCHAER, 2010)

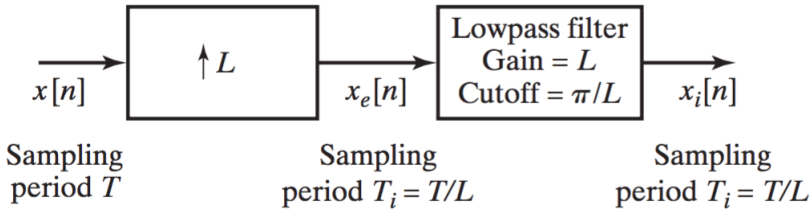
$$x_i[n] = x(nT_i) \tag{3.15}$$

e

$$x_i[n] = x[n/L] = x_c(nT/L), \qquad n = 0, \pm L, \pm 2L, \ldots \tag{3.16}$$

Para obter $x_i[n]$ em função de $x[n]$, utiliza-se o processo discreto no tempo definido como *expander*, conforme Figura 36, onde sua saída é dada por

Figura 36 – Sistema genérico para aumentar a taxa de amostragem em um fator L



Fonte: (OPPENHEIM; SCHAER, 2010)

$$x_e[n] = \begin{cases} x[n/L), & n = 0, \pm L, \pm 2L, \ldots, \\ 0, & \text{caso contrário,} \end{cases} \tag{3.17}$$

que por ser escrito como

$$x_e[n] = \sum_{k=-\infty}^{\infty} x[k]\delta[n - kL] \tag{3.18}$$

A transformada de Fourier para $x_e[n]$ pode ser expressa como

$$X_e(e^{jw}) = sum_{n=-\infty}^{\infty} \left(\sum_{k=-\infty}^{\infty} x[k]\delta[n - kL] \right) e^{-jwn} \tag{3.19}$$

$$= \sum_{k=-\infty}^{\infty} x[k]e^{-jwLk} \tag{3.20}$$

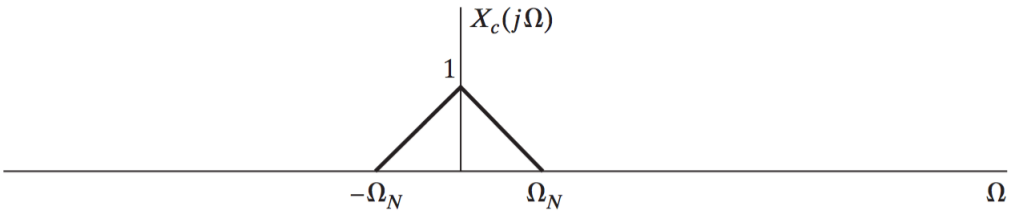
$$= X(^{-jwL}) \tag{3.21}$$

Para o filtro da Figura 36, sua equação é dada por

$$x_i[n] = \sum_{k=-\infty}^{\infty} x[k] \frac{sin[\pi(n - kL)/L]}{(n - kL)/L} \tag{3.22}$$

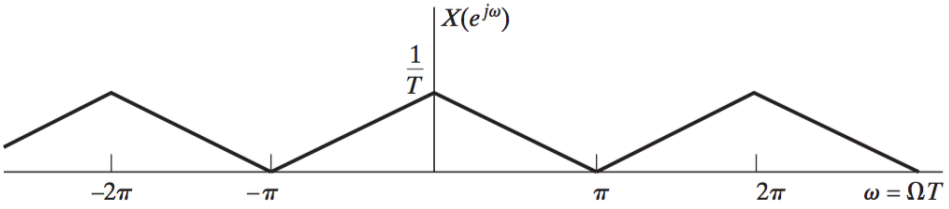
Da mesma forma que analisado para o caso da dizimação, seja $L = 2$, assim, temos os gráficos de $X_c(j\omega)$, $X(e^{jw})$, $X_e(e^{jw})$, $H_i(e^{jw})$ e $X_i(e^{jw})$, apresentados respectivamente nas Figuras 37, 38, 39, 40 e 41.

Figura 37 – DTFT do sinal $X_c(e^{jw})$



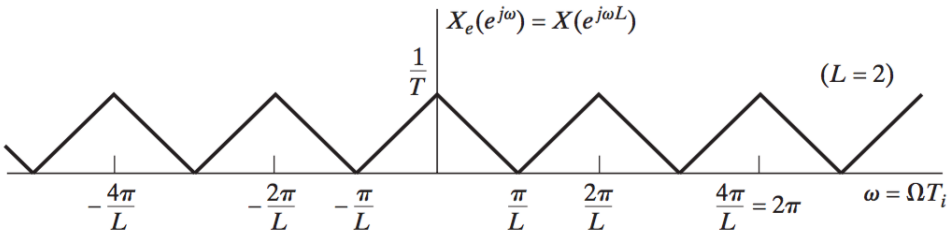
Fonte: (OPPENHEIM; SCHAER, 2010)

Figura 38 – DTFT de $X(e^{jw})$



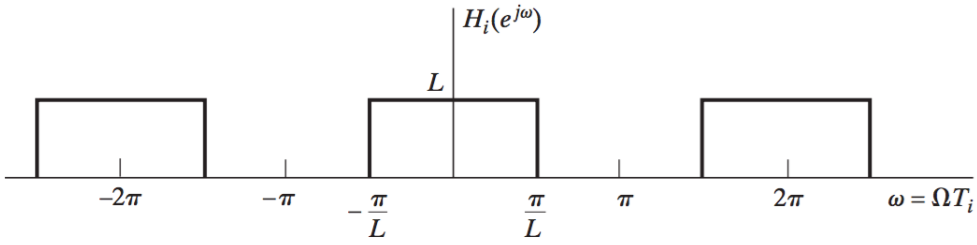
Fonte: (OPPENHEIM; SCHAER, 2010)

Figura 39 – DTFT de $X_e(e^{jw})$



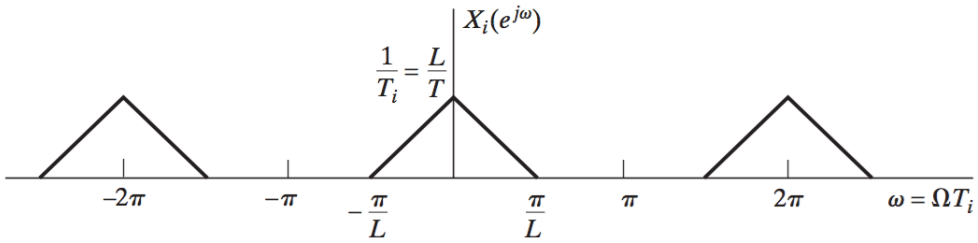
Fonte: (OPPENHEIM; SCHAER, 2010)

Figura 40 – DTFT de $H_i(e^{jw})$



Fonte: (OPPENHEIM; SCHAER, 2010)

Figura 41 – DTFT de $X_i(e^{jw})$

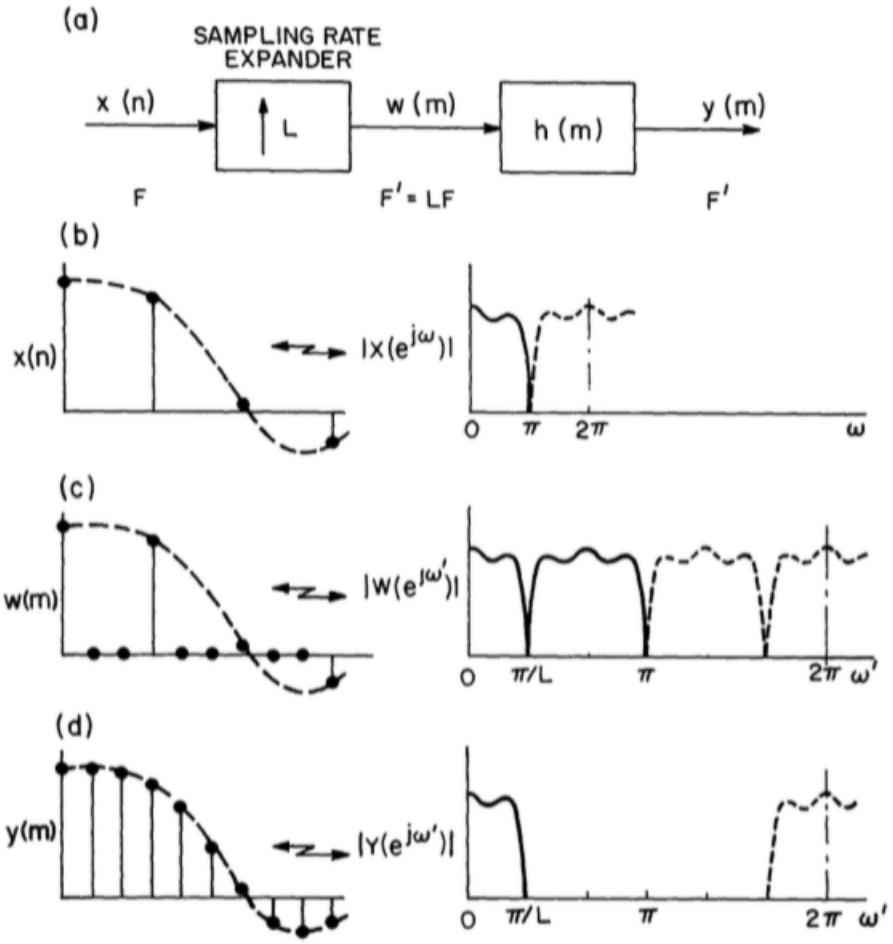


Fonte: (OPPENHEIM; SCHAER, 2010)

No caso do *upsampling* nota-se que o espectro de frequências é contraído pelo fator L em relação ao sinal original, diminuindo sua banda. Com a utilização do filtro, as cópias indesejadas são retiradas, assim, permanecem apenas as com periodicidade de 2π.

Na figura 42 há um caso típico de interpolação, onde utiliza-se o sistema apresentado no item (a). No item (b) apresenta-se o sinal originalmente amostrado. Em (c) ocorre o *zero-stuffing* e em (d) o sinal na saído do filtro $h(m)$. Também é possível visualizar a compressão, em relação ao sinal original, do espectro de frequências pelo mesmo fator L.

Figura 42 – Processo de interpolação por um fator L



Fonte: (CROCHIERE; RABINER, 1983)

3.3 Modulação de amplitude em quadratura - *quadrature amplitude modulation (QAM)*

Para se transmitir um sinal, geralmente um *stream* de *bits*, contendo algum tipo de informação é inevitável que ele passe por um canal de comunicação, isto é, a barreira física entre transmissor e receptor. Neste processo é possível que o sinal sofra modificações através de ruído, distorção, atenuação e interferência. O objetivo de se transmitir sinais é que seja possível receber essa mesma informação. Para que isto seja possível é necessário que este *stream* de dados sofra algumas modificações, isto é, operações com suas características originais, antes de ser transmitido, possibilitando, assim, uma tolerância às possíveis degradações que possa sofrer. O processo de mapear a sequência de dados digitais através do canal é chamado de modulação digital.

Tendo em vista a Equação 3.13, os sinais $x_i(t)$ e $x_q(t)$ podem assumir valores que componham uma conformidade entre si. Assim, abaixo serão exemplificados alguns destes casos específicos.

3.3.1 Modulação por deslocamento de fase - *Binary phase-shift keying* (BPSK)

Na modulação por deslocamento de fase o os sinais em quadratura podem ter sua fase alterada entre 0° e 180°. Nesta configuração $x_q(t) = 0$ e $x_i(t) = 1$ ou $x_i(t) = -1.$, assim:

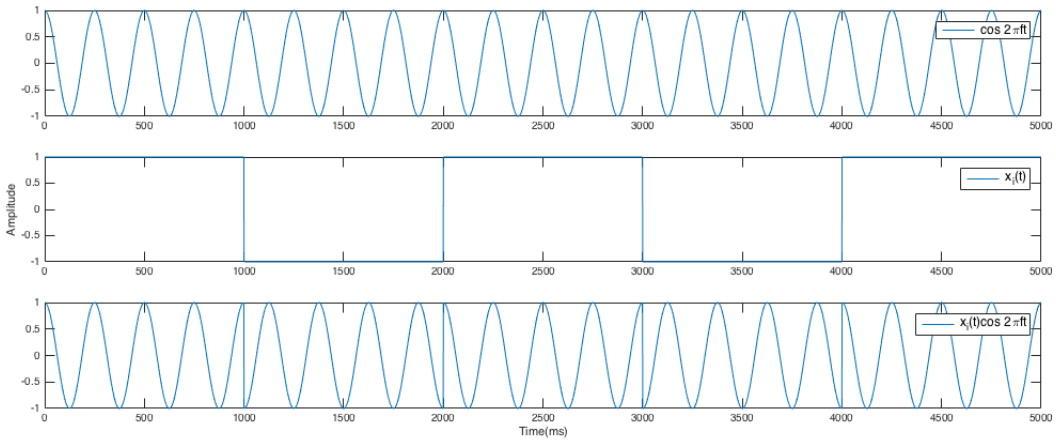
$$x_{BPSK_1}(t) = 1\cos 2\pi f_0 t - 0\sin 2\pi f_0 = \cos 2\pi f_0 t \tag{3.23}$$

$$x_{BPSK_2}(t) = -1\cos 2\pi f_0 t - 0\sin 2\pi f_0 = -\cos 2\pi f_0 t = \cos\left(2\pi f_0 t - \pi\right) \tag{3.24}$$

onde $x_{BPSK_1}(t)$ e $x_{BPSK_2}(t)$ representam os possíveis sinais para esta modulação.

Fazendo $2\pi f_0 = 25\,\text{ms}$ e $x_i(t)$ conforme apresentado no gráfico da Figura 43, nesta mesma figura são mostrados o sinal antes da modulação, $\cos 2\pi f_0 t$, e o sinal modulado, $x_i(t)\cos 2\pi f_0 t$.

Figura 43 – Modulação BPSK



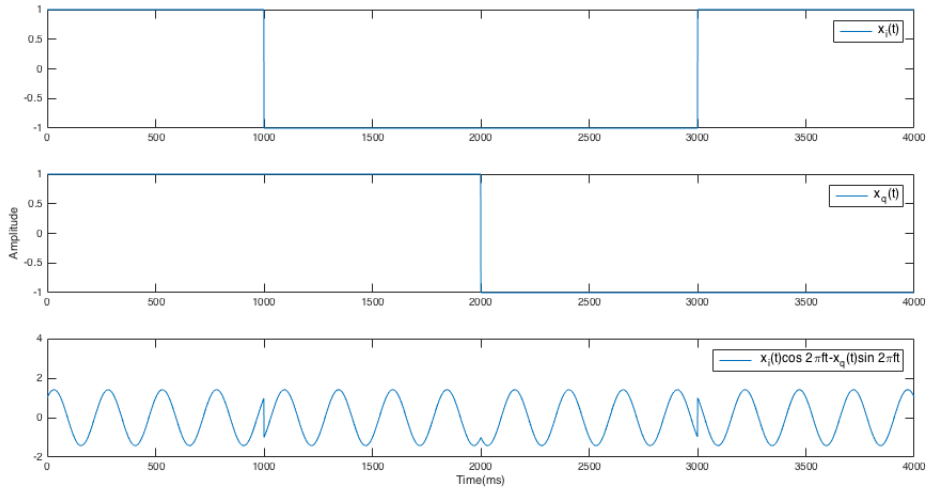
Fonte: Produzido pelo autor

3.3.2 Modulação por deslocamento de fase em quadratura *Offset quadrature phase-shift keying* - 4QAM

Assim como no caso anterior a componente em fase poderia assumir apenas dois valores, para este caso tanto a componente em fase como a componente em quadratura poderão assumir pares de valores para criar 4 possíveis deslocamentos de fase: 45°, 135°, 225° e 345°.

Para ilustrar melhor este comportamento, os gráficos da Figura 44 exibem a configuração deste tipo de modulação.

Figura 44 – Modulação 4QAM



Fonte: Produzido pelo autor

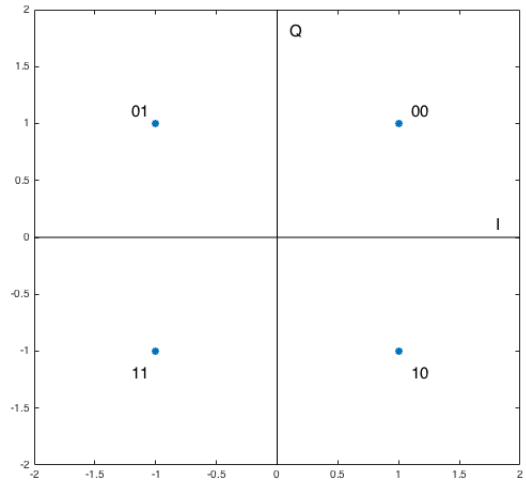
Na Tabela é possível identificar a relação entre as combinações dos sinais de fase e quadratura e os deslocamentos de fase para esta modulação.

Tabela 1 – Deslocamento de fase na modulação 4QAM

Sinal em fase $x_i(t)$	Sinal em quadratura de fase $x_q(t)$	Deslocamento de fase
1	1	45°
-1	1	135°
-1	-1	225°
1	-1	345°

Outra maneira de se visualizar a relação entre os módulos dos sinais em fase, I, e quadratura, Q, é através do diagrama de constelação. Representa-se no eixo vertical o valor da componente I e no eixo horizontal o valor da componente Q. O ângulo entre o vetor resultante das duas componentes como eixo horizontal, com referência anti-horária, corresponde ao deslocamento de fase para este par de sinais. Na Figura 45 é mostrado o diagrama de constelação da modulação 4QAM.

Figura 45 – Diagrama de constelaçãopara a modulação 4QAM



Fonte: Produzido pelo autor

Assim, as duas modulações apresentadas serviram de exemplo de como é feita a modulação de amplitude em quadratura de fase. No entanto, estas combinações para os sinais I e Q não se limitam a estes casos, podendo assumir outros valores. Em geral, podem assumir a forma NQAM, onde N é o número que está relacionado com a quantidade de diferentes pontos que podem ser transmitidos, isto é, o número máximo de valores distintos de informações, fase e amplitude, compostos pelos sinais em fase e quadratura. Para o caso BPSK, $N = 2$, pois há somente duas possibilidades de informações a serem transmitidas. Para 4QAM há quatro possibilidades, um mesmo valor de módulo, mas com 4 possibilidades de ângulo.

Os pares de valores assumidos por I e Q (módulo e fase) representam um símbolo, onde geralmente são uma sequência de 0's e 1's, *stream* de *bits*. Esta sequência tem como objetivo representam um número. Logo, o tamanho deste símbolo, isto é, quantidade de *bits* que o compõe está relacionado com o tamanho da constelação de acordo com a Equação 3.25.

$$n = \sqrt{N} \tag{3.25}$$

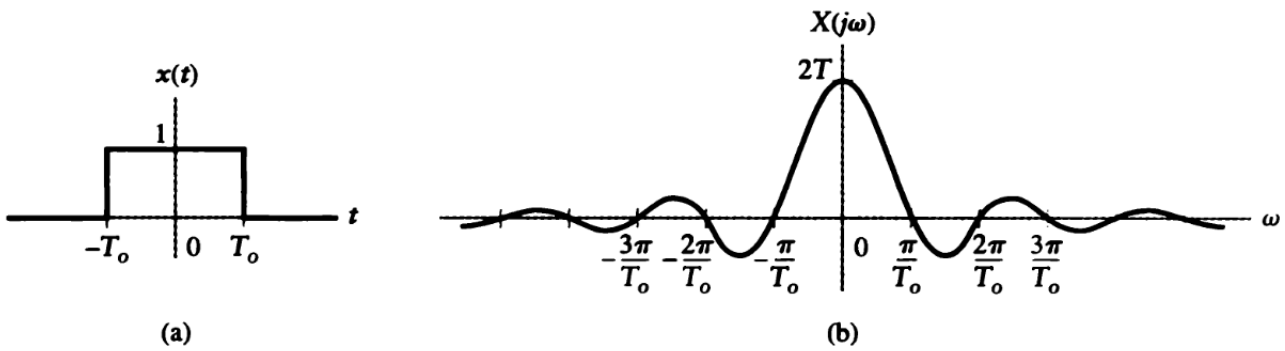
Desta maneira, é possível verificar na Figura 45 que o tamanho do símbolo que pode ser transferido pela modulação 4QAM é 2, conforme 3.25.

3.4 Modelagem de pulso - *Pulse shaping*

Na modulação QAM a escolha dos valores dos pares dos sinas IQ varia de acordo com o símbolo que será transmitido. Assim, o valor resultante para a transmissão (modulação digital), i.e, o valor instantâneo de 3.13, será transmitido em forma de pulsos, pois trata-se de um sistema discreto. A escolha do formato desses pode determinar a robustez da transmissão da mensagem.

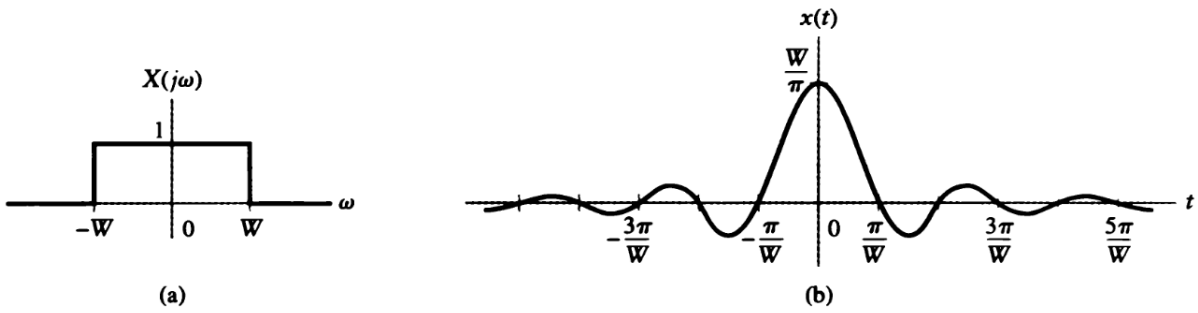
A escolha do tipo de pulso a ser utilizada determinará qual o conjunto de frequências para se gerar este sinal, ou seja, qual será a banda necessária. A análise para a escolha do pulso pode ser restringida analisando-se dois casos extremos: um pulso retangular, Figura 46, e um pulso *sinc*, Figura 47.

Figura 46 – Pulso retangular no domínio do tempo (a) e no domínio da frequência (b)



Fonte: (HAYKIN; VEEN, 2003)

Figura 47 – Pulso sinc no domínio da frequência (a) e no domínio do tempo (b)

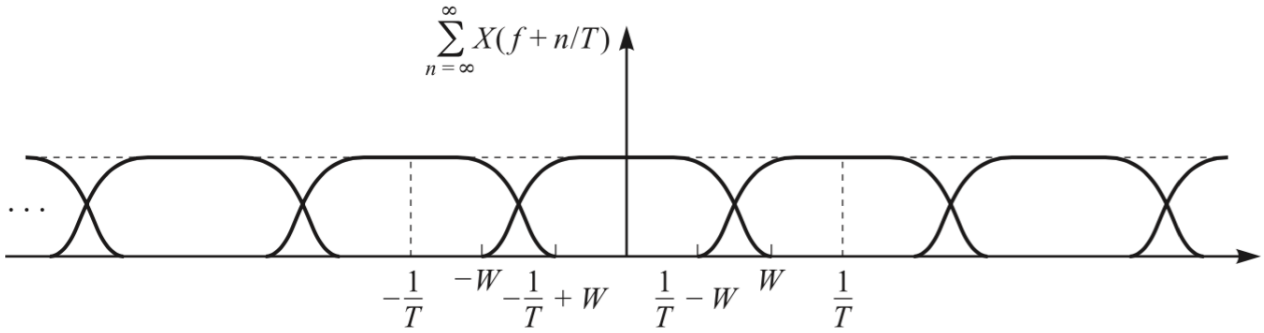


Fonte: (HAYKIN; VEEN, 2003)

Nota-se que o pulso retangular é muito bem definido no tempo comparado ao

pulso *sinc*, no entanto, a banda necessária para representar este pulso, é, teoricamente, infinita, como se pode ver no item (b) da Figura 46. Assim sendo, torna-se impossível sua implementação, pois em um sistema de comunicação há limitação de banda. Já para o pulso *sinc*, seu espectro de frequências é bem determinado e finito, item (a), porém, sua representacão temporal possui lóbulos fora do intervalo do pulso principal, lóbulo central. Quando se transmite uma sequência de pulsos, estes lóbulos secundários podem interferir interferir entre si, causando assim a interferência intersimbólica, *intersymbol interference* (ISI), Figura 48.

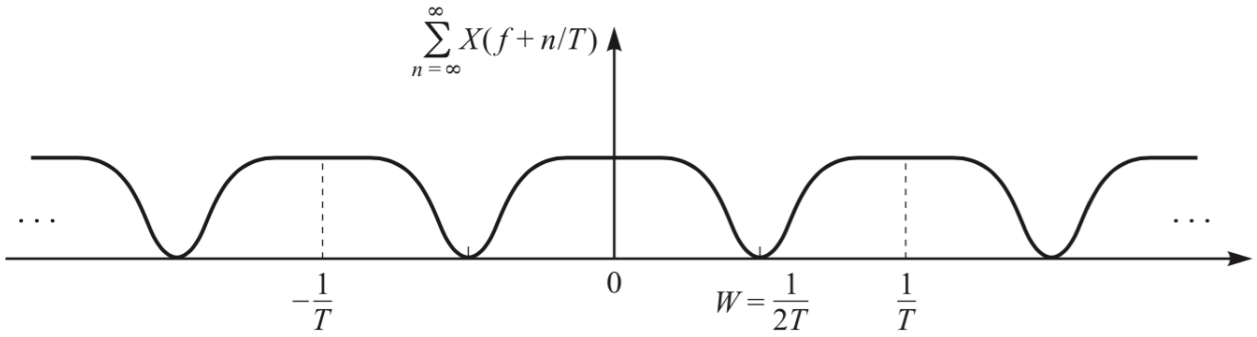
Figura 48 – Interferência entre pulsos, cruzamento das linhas (ISI)



Fonte: (PROAKIS, 2008)

Para que seja possível realizar a transmissão de uma sequência de pulsos que possam ser recuperados na recepção, é necessário modelar o tipo de pulso que será transmitido, de maneira que quando o sinal seja amostrado no receptor, não haja a interferência destes pulsos nos tempos de amostragem. Essa modelagem está relacionada com o critério de uma transmissão sem distorção, estudada por Nyquist em 1928, na teoria de transmissão telegráfica, que continua sendo válida até hoje (HAYKIN, 2014). A partir destes estudos, Nyquist definiu alguns critérios, Nyquist's *criterion*, para as características destes pulsos. Dentre eles a banda de Nyquist, *Nyquist bandwidth*, $W = R/2 = 1/2T$, onde R representa a banda dos símbolos transmitidos. No entanto, sua implementação ainda não é realizável, pois o pulso com tais características inicia em $-\infty$, logo, para gerá-lo deve ser esperado um tempo infinito para que ocorra; qualquer tentativa de tentar truncá-lo aumentaria a banda além de R (LATHI, 1998). Para tornar a implementação destes pulsos possível, a estratégia adotada é a de extender a banda mínima para um valor ajustável entre W e $2W$. Assim, uma modelagem de pulsos que cumpre todos os critérios de Nyquist e possui a banda citada acima é o chamado *raised-cosine spectrum*, Figura 49.

Figura 49 – Eliminação da ISI através da escolha de uma modelagem de pulso que atende aos critérios de Nyquist



Fonte: (PROAKIS, 2008)

De acordo com (PROAKIS, 2008), as equações definem o pulso *raised cosine* no domínio da frequência e do tempo são dadas por

$$X(f) = \begin{cases} T & 0 \leq |f| \leq \dfrac{1-\beta}{2T} \\ \dfrac{T}{2}\left\{1 + cos\left[\dfrac{\pi T}{\beta}\left(|f| - \dfrac{1-\beta}{2T}\right)\right]\right\} & \dfrac{1-\beta}{2T} \leq |f| \leq \dfrac{1+\beta}{2T} \\ 0 & |f| > \dfrac{1+\beta}{2T} \end{cases} \tag{3.26}$$

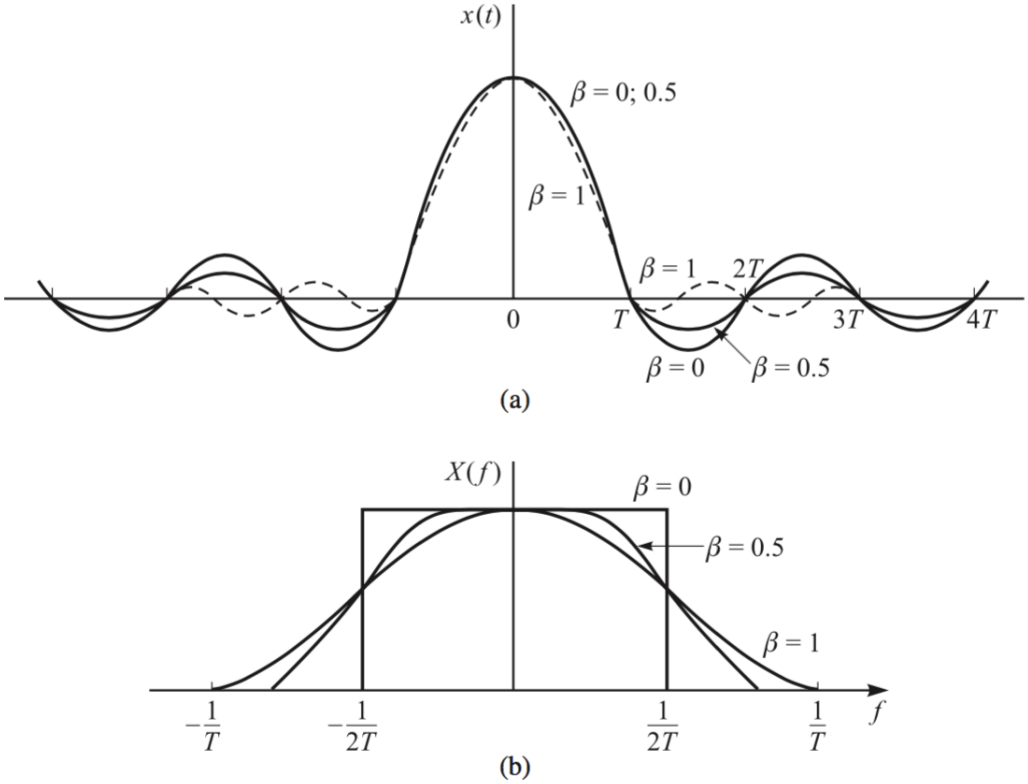
$$x(t) = sinc(\pi t/T)\frac{cos(\pi\beta t/T)}{1 - 4\beta^2 t^2/T} \tag{3.27}$$

onde β é chamado de *roll-off factor* e pode estar dentro do intervalo $0 \leq \beta \leq 1$. Ele indica o excesso de banda utilizado sobre o caso ideal. Assim, a nova banda de transmissão será

$$B_T = W(1+\beta) \tag{3.28}$$

No gráfico apresentado na Figura 50 são apresentados os casos para $\beta = 0$, $\beta = 0,5$ e $\beta = 1$, que correspondem aos casos de 0%, 50% e 100% de excesso de banda.

Figura 50 – Pulsos com o formato *raised cosine spectrum*. (a) Domínio do tempo. (b) Domínio da frequência



Fonte: (PROAKIS, 2008)

3.5 Transformada de Hilbert

Como visto na seção 2.1, a transformada de Hilbert é necessária para a implementação da modulação IQ, pois é através dela que a parte imaginária do envelope complexo do sinal a ser transmitido é recuperado, conforme 3.8. Para melhor compreender do porquê de sua utilização e qual propriedade a torna capaz de relacionar as partes real imaginária do sinal transmitido, primeiro, será feita sua definição e posteriormente indicada a propriedade a ser utilizada.

3.5.1 Definição

De acordo com (HAHN, 1996), qualquer par unidimensional integral pode ser escrito como:

$$u(t) \Longleftrightarrow U(s) \tag{3.29}$$

O par de integrais que definem a transformada de Hilbert é dado por:

$$U(s) = \frac{1}{\pi}\int_{-\infty}^{\infty}\frac{u(t)}{s-t}dt; -\infty < t < \infty \tag{3.30}$$

$$u(t) = \frac{1}{\pi}\int_{-\infty}^{\infty}\frac{U(s)}{t-s}dt; -\infty < t < \infty \tag{3.31}$$

onde frequentemente são escritas em termos da notação de convolução

$$v(t) = \frac{1}{\pi t} \star u(t) \tag{3.32}$$

$$u(t) = -\frac{1}{\pi t} \star v(t) \tag{3.33}$$

logo, $v(t)$ e $u(t)$ formam um par transformado de Hilbert.

3.5.2 Interação com a transformada de Fourier

A função $\frac{1}{\pi t}$ tem sua transformada de Fourier sendo

$$-jsgn(f) = \begin{cases} -j, & \text{se } f > 0 \\ 0, & \text{se } f = 0 \\ j, & \text{se } f < 0 \end{cases} \tag{3.34}$$

Seja $U(f)$ a transformada de Fourier para $u(t)$, a convolução da Equação 3.33 é dada por

$$V(f) = -jsgn(f)U(f). \tag{3.35}$$

A compreensão da transformada de Hilbert pode ser melhor entendida através da análise domínio da frequência do que no domínio do tempo. Através da Equação 3.35 é possível verificar que a magnitude de $U(f)$ não é alterada. No entanto, valores da transformada de Fourier para frequências positivas são multiplicados por $-j$, o equivalente a uma mudança de fase de 90° e frequências negativas multiplicadas por j, equivalente a uma mudança de fase de $-90°$.

Suponha um sinal $U(f) = a + jb$ para algum valor de frequência f, assim $V(f) = b - ja$ se $f > 0$ e $V(f) = -b + ja$ se $f < 0$. Logo, a transformada de Hilbert simplesmente troca os valores real e imaginário de $U(f)$ enquanto modifica o sinal de um deles.

Retornando à equação 3.13, apresentada anteiormente, aplicando a transformada de Hilbert

$$\mathscr{H}[x(t)] = \mathscr{H}[x_i(t)\cos 2\pi f_0 t - x_q(t)\sin 2\pi f_0] \tag{3.36}$$

como a transformada de Hilbert possui a propriedade de ser linear, podemos escrever

$$\mathscr{H}[x(t)] = \mathscr{H}[x_i(t)\cos 2\pi f_0 t] - \mathscr{H}[x_q(t)\sin 2\pi f_0]$$

como $x_i(t)$ e $x_q(t)$ são sinais da banda base e não possuem o espectro de frequência em sobreposição às funções trigonométricas

$$\mathscr{H}[x(t)] = x_i(t)\mathscr{H}[\cos 2\pi f_0 t] - x_q(t)\mathscr{H}[\sin 2\pi f_0]$$

assim, substituindo o resultado da transformada para as funções trigonométricas

$$\begin{aligned} \mathscr{H}[x(t)] &= x_i(t)\sin 2\pi f_0 - x_q(t)(-\cos 2\pi f_0) \\ \mathscr{H}[x(t)] &= x_i(t)\sin 2\pi f_0 + x_q(t)\cos 2\pi f_0 \end{aligned} \tag{3.37}$$

sendo $\mathscr{H}[x(t)] = \hat{x}(t)$. A Equação 3.37 é idêntica à Equação 3.14.

3.6 Dizimação - *Decimation* e *Downsampling*

Decimation ou Dizimação é o processo de reduzir a taxa de amostragem. Para que isso seja feito, normalmente filtra-se o sinal com um filtro passa-baixas descartando-se posteriormente algumas de suas amostras. *Downsampling* é o processo de somente descartar as amostras, sem que o processo de filtragem seja necessário.

Duas são as motivações para realizar o *downsampling* de um sinal: reduzir a frequência de amostragem da saída que esteja ligado à entrada de outro operando em menor frequência; diminuir o custo de programação, num contexto de processamento de sinais digitais. Geralmente todas operações e memória de programa estão proporcionalmente ligadas à frequência de amostragem, assim, diminuindo-a, pode-se diminuir os custos de implementação do sistema.

Para que não haja nenhuma perda de informação do sinal ao passar porum *downsampling*, é necessário garantir que o sinal tenha sido amostrado a uma taxa superior àquela exigida pelo critério de Nyquist, referenciada a nova taxa de amostragem do sistema, que remete a dizimação.

Seja $x_c(t)$ um sinal contínuo no tempo; sua representacão discreta $x[n] = x_c(nT)$, onde T representa o período de amostragem. Seja $T_d = TM$ o novo período de amostragem, onde M é um número inteiro, logo $T_d > T$. Assim, de acordo com (OPPENHEIM; SCHAER, 2010), as equações para a transformada de Fourier para o tempo discreto, *discrete-time Fourier transform* (DTFT), para o sinal amostrado em um período T e para peíoro T_d, são, respectivamente

$$X(e^{jw}) = \frac{1}{T}\sum_{k=-\infty}^{\infty} X_c\left[j\left(\frac{w}{T} - \frac{2\pi k}{T}\right)\right] \tag{3.38}$$

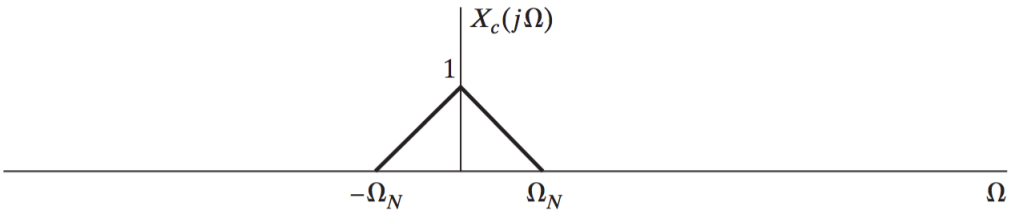
$$X_d(e^{jw}) = \frac{1}{MT}\sum_{r=-\infty}^{\infty} X_c\left[j\left(\frac{w}{MT} - \frac{2\pi r}{MT}\right)\right] \tag{3.39}$$

onde $r = i + kM$. A Equação 3.38 representa a DTFT do sinal discreto, amostrado em T, em função do sinal contínuo e a Equação 3.39 representa a DTFT do sinal com o perído de amostragem T_d em termos do sinal contínuo. Desta maneira, é possível relacionar $X(e^{jw})$ e $X_d(e^{jw})$

$$X_d(e^{jw}) = \frac{1}{M} \sum_{i=0}^{M-1} X\left(e^{j(w/M - 2\pi i/M)}\right) \tag{3.40}$$

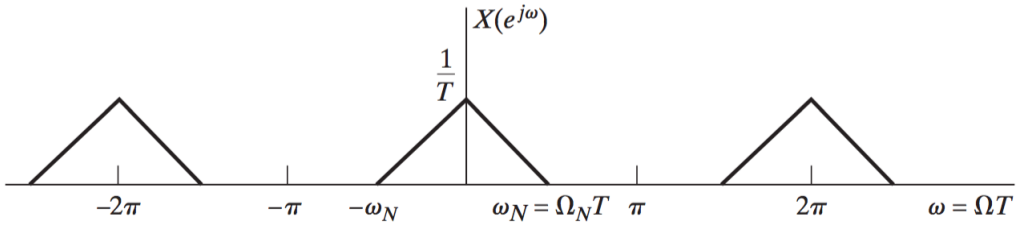
Sendo ω_N a frequência máxima do sinal conítnuo, para não haver caso de *aliasing*, é necessário que $2\pi/T = 4\omega_N$. Supondo $M = 2$, os gráficos para os sinais $X_c(e^{jw})$, $X(e^{jw})$ e $X_d(e^{jw})$, são mostrados nas Figuras 51, 52 e 53.

Figura 51 – DTFT do sinal $X_c(e^{jw})$



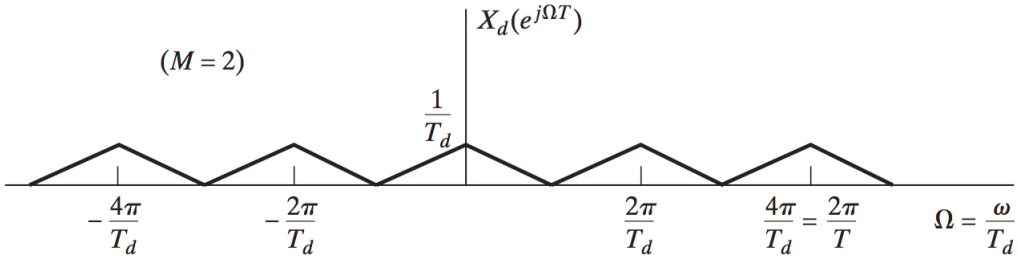
Fonte: (OPPENHEIM; SCHAER, 2010)

Figura 52 – DTFT do sinal $X(e^{jw})$



Fonte: (OPPENHEIM; SCHAER, 2010)

Figura 53 – DTFT do sinal $X_d(e^{jw})$



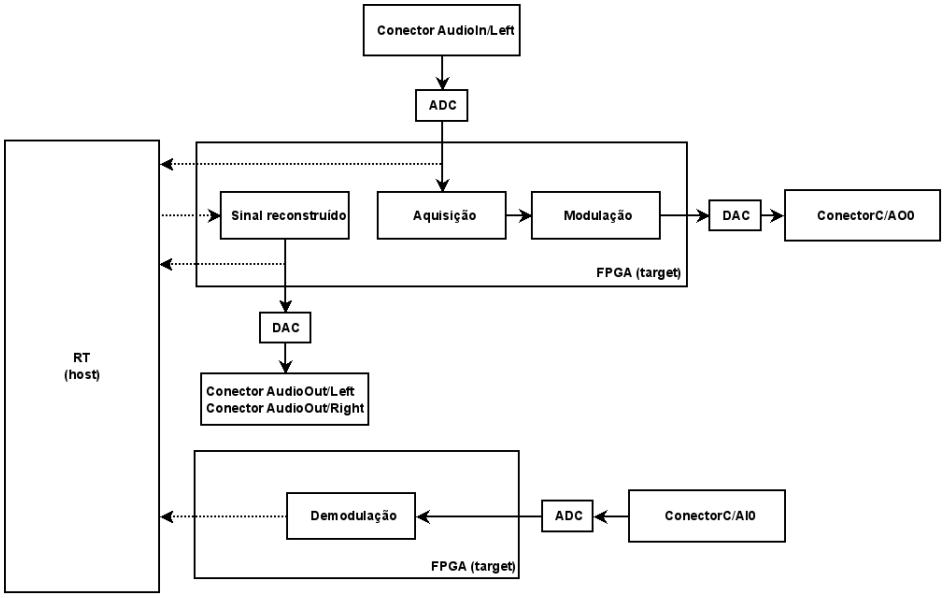
Fonte: (OPPENHEIM; SCHAER, 2010)

A partir das figuras nota-se que o espectro de frequências do sinal amostrado possui várias cópias com uma periodicidade de 2π ao longo do eixo w, que corresponde ao processo de *sampling*. Assim, ao fazer a dizimação do sinal amostrado o espectro de frequências acaba sendo expandido, ocupando todos os espaços adjacentes entre as antes cópias provenientes da discretização.

4 Implementação em FPGA e RT

O sistema implementado é constituído de um conjunto de rotinas desenvolvidas em LabVIEW e embarcadas, parte na plataforma FPGA, parte no sistema operacional Linux RT do myRIO. Como se pode ver na Figura 54 os módulos referentes à aquisição dos sinais, modulação, transmissão, recepção e demodulação. Entre as entradas e saídas analógicas, tanto as de áudio quanto as do conector C, há conversores analógicos digitais (ADC) e digitais analógicos (DAC), e o FPGA, também representados na figura.

Figura 54 – SDR completo - Conexões FPGA - RT



Fonte: Produzido pelo autor

A seguir serão descritas detalhadamente cada uma dessas rotinas, separadas nas que acontecem no FPGA e no RT.

4.1 FPGA

Na porção correspondente ao FPGA encontram-se as seguintes processos presentes nos laços descritos abaixo:

- “acquisition loop”: loop de aquisição de dados;
- “modulation loop”: mapeamento, filtragem e *oversampling*;

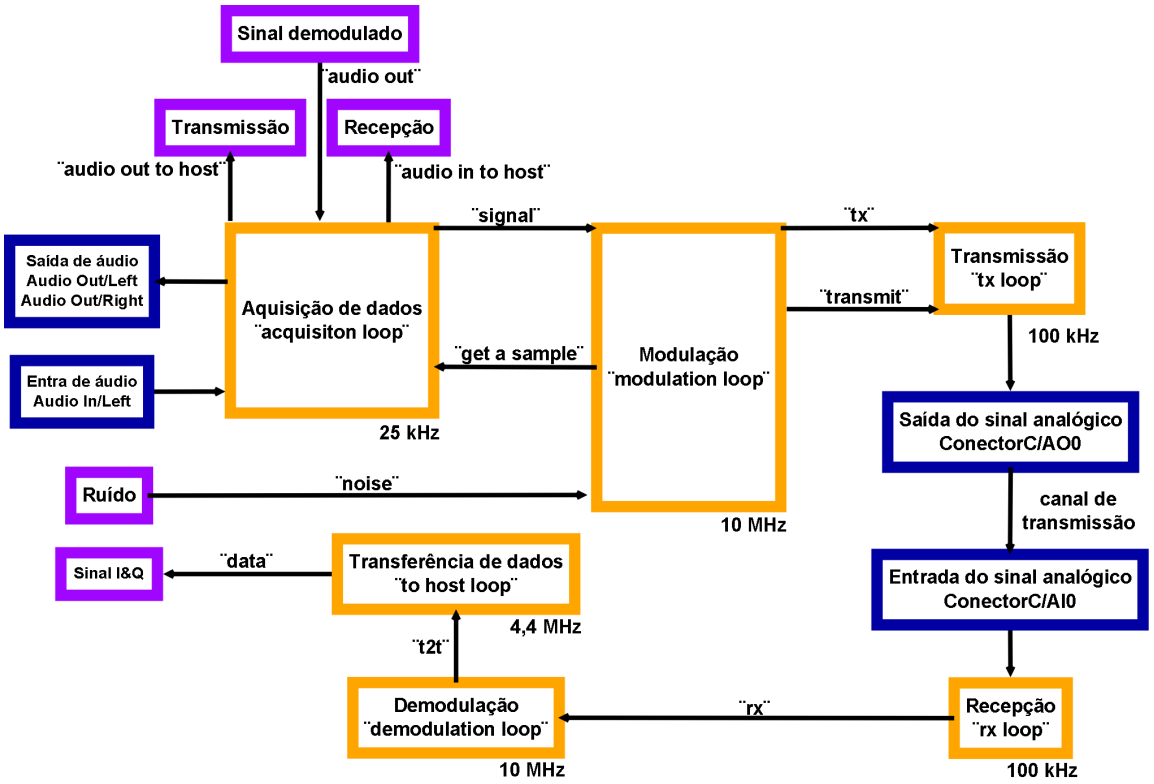
- "tx loop": transmissão;
- "rx loop": recepção;
- "demodulation loop": recuperação da parte imaginária do sinal transmitido, demodulação e *downsampling*;
- "to host loop": transferência dos dados para o *host*;

Todos estes processos operam paralelos e sincronizados, pois todos encontram-se com algum *clock* de referência presentes na mesma placa onde encontra-se o FPGA. No caso dos loops de modulação e demodulação, ambos operam com um *clock* de 10 MHz. Os *loops* de aquisição, transmissão e recepção estão sincronizados com o *loops* de modulação e operam a frequências determinadas por este *loop*. O *loop* de demodulação está sincronizado com o de modulação ao utilizar o mesmo *clock* de referência. Já o *loop* de transferência de dados pode ter uma frequência de operação variável dependendo de como se comporta o canal DMA da FIFO "data", no entanto, deve operar o mais rápido possível, tendo como limitações de velocidade o preenchimento dos valores no loop de demodulação e a escrita dos dados pela utilização de um *loop for*.

O sincronismo dos dados de transmissão e recepção é feito para que não aconteça escrita redundante de dados evitando que o *loop* de recepção receba repetidas vezes os valores transmitidos. Assim se garante que os dados estejam alinhados. Tratando-se de uma simulação de um canal de transmissão, fica definido, para esta situação, que está disponível a informação da frequência do sinal de transmissão. Desta maneira, é possível utilizar esta estrutura de "loopback test", onde se escreve um sinal em uma porta de saída e lê-se o mesmo em outra, ambas as portas estando sincronizadas nos tempos de escrita e leitura.

O funcionamento do sistema presente no FPGA e sua relação com os demais pode ser expresso pelo fluxograma da Figura 55.

Figura 55 – Fluxo de dados entre o FPGA, saídas e entradas e RT. Na cor roxa se encontram as variáveis entre RT e FPGA. Em laranja, os processos que ocorrem somente no FPGA, com a respectiva frequência. Em azul, os conectores analógicos

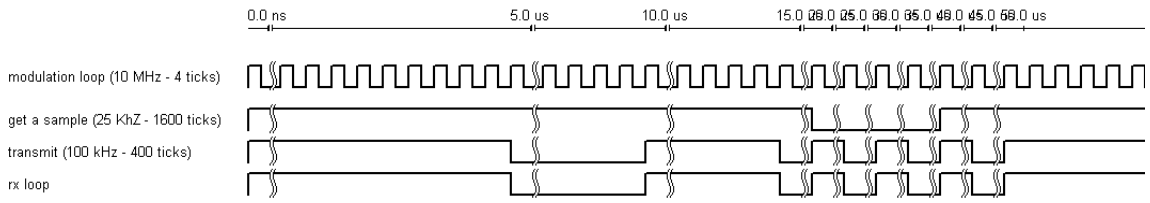


Fonte: Produzido pelo autor

4.1.1 Diagramas de tempo

A relação entre os tempos de execução é dada pela relação mostrada na Figura 56.

Figura 56 – Relação entre os *clocks* de cada *loop*



Fonte: Produzido pelo autor

Onde, tem-se o período do *loop* de aquisição com uma duração de 1600 *ticks*, controlado através da FIFO "transmit", e o *loop* de recepção, com duração de 400 *ticks*,

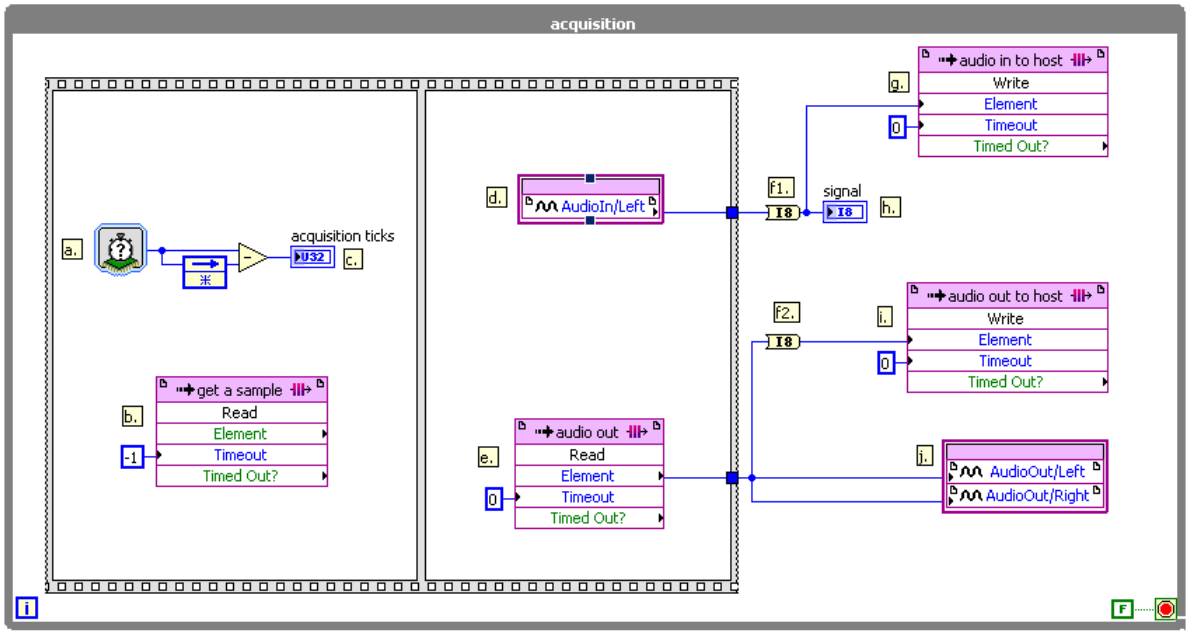
sincronizado com o de transmissão. Na Figura 56 foram feitas compressões temporais para a melhor visualização entre a relação dos *ticks* de cada laço.

Para o *loop* de modulação, “modulation loop”, implementou-se uma máquina de estados para que fosse possível utilizar as otimizações das implementações lógicas permitidas para um *loop* SCTL, mesmo não fazendo-se o uso de cada iteração para diferentes operações. Desta maneira, realizam-se as operações desejadas para com o sinal em duas combinações de estados, “read sample” seguido de “hold” e “write zero” seguido de “hold”, e após estas combinações o *loop* permanece no estado “hold” por 98 iterações. Com isso se tem um controle da frequência de transmissão do sinal, pela limitação da frequência de escrita do dado no conversor D/A no conector C, limitado em 345 kHz. Para a escrita no dado no conversor A/D no mesmo conector sua frequência é de 500 kHz . Na presente implementação foi escolhida a frequência de 100 kHz para a transmissão, no entanto, é possível que se aumente esta frequência, conforme o limites entre A/D e D/A mencionados anteriormente. Assim, o diagrama temporal da máquina de estados é mostrado na Figura 68.

A implementação no FPGA é composta por um conjunto de processos paralelos, cada qual formando um módulo com função específica. Como cada processo paralelo corresponde a uma estrutura de laço, serão descritos os *loops* utilizados para cada função do sistema.

4.1.2 Aquisição e escrita dos valores da entrada e saída analógica dos conectores de áudio do myRIO

Figura 57 – Diagrama de blocos do *loop acquisition*



Fonte: Produzido pelo autor

a. Sub VI "Tick Count": Contagem das ocorrências de aquisição.

b. FIFO "get a sample": Controle de execução do *loop* para ler a variável local "signal", item h.. Sub VI "Tick Count": Contagem das ocorrências de aquisição.

c. Indicador numérico "acquisition ticks": Exibe a diferença de tempo entre duas iterações.

d. Conector de entrada AudioIn/Left: Entrega o valor colhido pela entrada analógica de áudio ao indicar numérico "signal".

e. FIFO "audio out": Recebe os dados demodulados e mapeados do RT.

f1. e f2. Conversores para variável numérica de 8 *bits*: Fazem a conversão de uma variável de 16 *bits* para 8 *bits*.

g. FIFO "audio in to host": Recebe os valores lidos pela entrada analógica de áudio para transferência ao RT.

h. Indicador numérico da variável local "signal": Exibe o valor recebido pela entrada analógica.

i. FIFO “audio out to host”: Recebe os valores lidos pela entrada analógica de áudio para transferência ao RT.

j. Conectores de saída Audio Out/Left e Audio Out/Right: Recebem os valores transferidos do RT ao FPGA pela FIFO “audio out” e os enviam para as saídas de audio.

4.1.2.1 Objetivo

Aquisição e escrita dos valores da entrada e saída analógica, respectivamente, dos conectores de áudio do myRIO. Os dois valores são transferidos através de duas FIFO's distintas, uma para cada sinal, ao RT para o cálculo da transformada rápida de Fourier, *fast Fourier transform* - FFT, e da transformada de Fourier de curto termo, *short-time Fourier transform* - STFT. Os dados de entrada são escritos em uma variável local (*signa*l), indicador numérico, para leitura no *loop* de modulação. A frequência de aquisição deste *loop* é controlada pelo *loop* de modulação. Na Figura 57 há a indicação dos elementos pertencentes a este *loop*.

4.1.2.2 Fluxo de dados

Dados de entrada:

- Conector AudioIn/Left, variável do tipo I16, inteiro de 16 *bits* (-32758 a 32767);
- FIFO “get a sample”, *dummy signal* booleano (verdadeiro ou falso);
- FIFO “audio out”, variável do tipo I16.

Dados de saída:

- Indicador numérico “acquisition ticks”, variável do tipo U32, inteiro de 32 *bits* (0 a 4294967295);
- Indicador numérico “signal”, variável do tipo I8, inteiro de 8 *bits* (-128 a 127);
- Conectores Audio Out/Left e Audio Out/Right, variáveis do tipo I16;
- FIFO's “audio in to host” e “audio out to host”, variáveis do tipo I8.

4.1.2.3 Tempo de execução

O tempo de execução é fixo em 40 µs por iteração. O controle de seu período é feito através da FIFO “get a sample”, que foi utilizada para exercer a mesma função como se fosse uma função de ocorrência. A limitação das funções de ocorrência é de não poderem ser utilizadas em diferentes domínios de *clock*. Seu controle é feito pelo *loop* de modulação, onde é feita a escrita desta “variável”. Sua configuração foi feita deixando

seu tempo de timeout em -1, onde a FIFO espera indefinidamente para ler um novo dado. Isto significa que a leitura da variável *dummy* só será feita quando seu valor for atualizado. Enquanto não houver um novo valor para a leitura, a FIFO ficará em estado *idle*. Através do indicador "acquisition ticks" é possível verificar a quantidade de *ticks* entre uma iteração e outra. Seu valor é de 1600 *ticks*, onde seu *clock* de referência é de 40 MHz (1 *tick* corresponde a 25 ns), ou seja, 25 kHz.

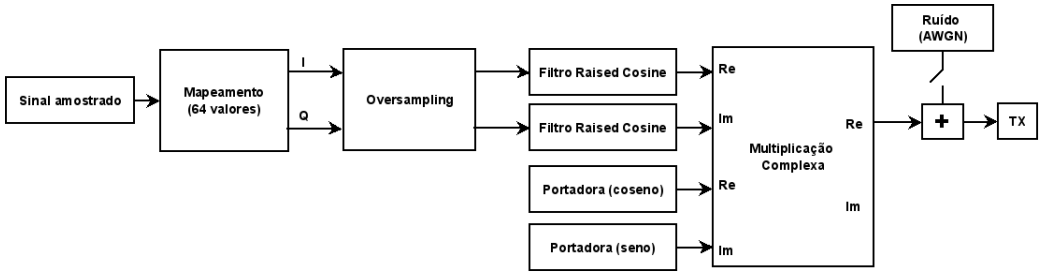
4.1.2.4 Descrição do funcionamento

Através do "sinal de *clock*" gerado pelo *loop* de modulação, este *loop* operada a uma taxa de 25 kHz. Assim, o sinal de áudio de entrada é adquirido com esta frequência de aquisição . O mesmo ocorrendo com o sinal de áudio da saída, ambos presentes no mesmo *loop*. Para o correto controle deste tempo de iteração, utiliza-se uma estrutura de programação *flat sequence*, no primeiro *frame* foi colocado a leitura da FIFO "get a sample". Neste mesmo *frame* está presente a sub VI "Tick Count" para a conferência da quantidade de *ticks* que são contados entre iterações. No segundo *frame* encontram-se o conector da entrada do sinal de áudio e a FIFO de leitura "audio out". O sinal de entrada é convertido em uma variável de 8 *bits* "signal" que será utilizada pelo *loop* de modulação para aquisição das amostras colhidas. Após ser convertido, esse dado é colocado na FIFO "audio in to host", que transfere estes dados para o RT. A FIFO "audio out" recebe os valores numéricos decorrentes dos dados demodulados após serem mapeados corretamente. Estes são escritos nos conectores de saída do sinal de áudio. O dado recebido por esta última FIFO, "audio out to host", é também convertido para um inteiro de 8 *bits* para ser transferido ao RT. Ambas transferências do sinal de entrada e saída são necessários para se realizar a FFT e a STFT do sinal, utilizadas para a visualização do espectro de frequência e o *waterfall plot*.

4.1.3 Modulação dos sinais coletados pelo *loop* de aquisição - *modulation loop*

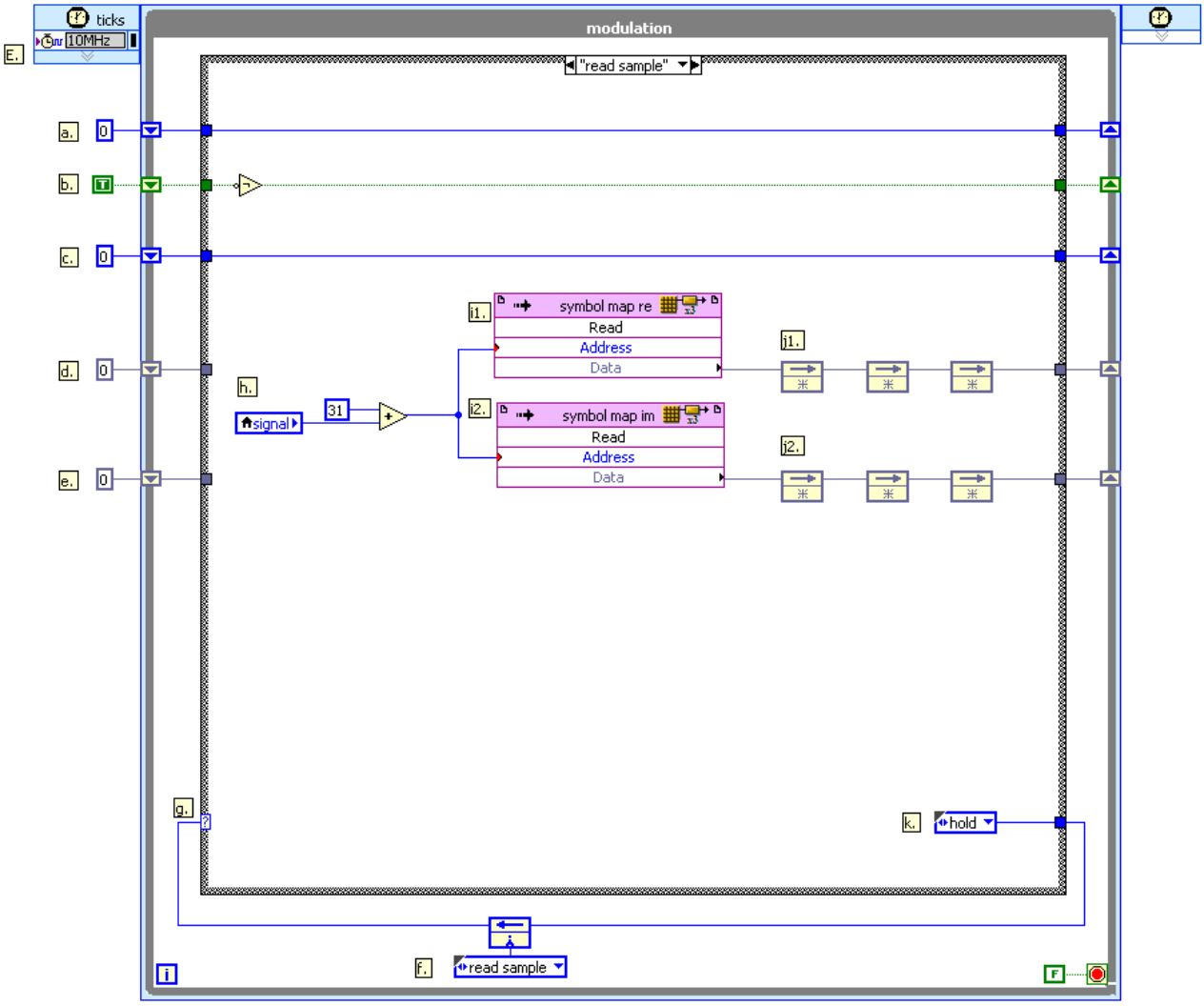
O processo de modulação que ocorre nesta parte do código pode ser descrito de acordo com o diagrama de contexto da Figura 58. O sinal de áudio amostrado é mapeado, isto é, para cada valor inteiro recebido do conversor AD é escolhido um par de valores para os sinais I e Q. Para o correto funcionamento do filtro, os sinais recebem inserções de três zeros entre seus valores amostrados (*oversampling*) e em seguida são filtrados, utilizando o filtro FIR *Raised Cosine*. Após, são multiplicados com os valores de seno e coseno da portadora. Utiliza-se a parte real da multiplicação, o resultado é um número complexo, e transmite-se, com a adição ou não de ruído gaussiano do tipo branco, o sinal modulado.

Figura 58 – Modulação



Fonte: Produzido pelo autor

Figura 59 – Diagrama de blocos do caso “read sample” do *loop* de modulação



Fonte: Produzido pelo autor

a. Constante inicial numérica: Valor inicial "0", conectada ao *shift register* que controla o número de vezes em que zeros foram inseridos entre os valores amostrados do sinal de entrada. Responsável pelo *oversampling*.

b. Constante inicial booleana: Valor inicial "F", conectada ao *shift register* que controla a escolha ou de um valor amostrado ou de um zero para o *oversampling* para ser encaminhado ao filtro FIR *Raised Cosine*.

c. Constante inicial numérica: Valor inicial "0", conectada ao *shift register* que controla o número de ciclos de *clock* em que o estado "wait" permanece.

d. e e. Constantes iniciais numéricas do tipo ponto fixo: Valores iniciais "0", conectadas aos *shift registers* responsáveis por acumular os valores dos dados que serão lidos no estado "hold".

f. *Feedback Node* com valor inicial do tipo *enum*: Estado inicial "read sample". Acumula o próxima estado a ser executado na máquina de estados.

g. Seletor de estados da máquina de estados: Recebe os valores numéricos descritos em **f.**.

h. Variável local "signal": Recebe o valor colhido no *loop* de aquisição.

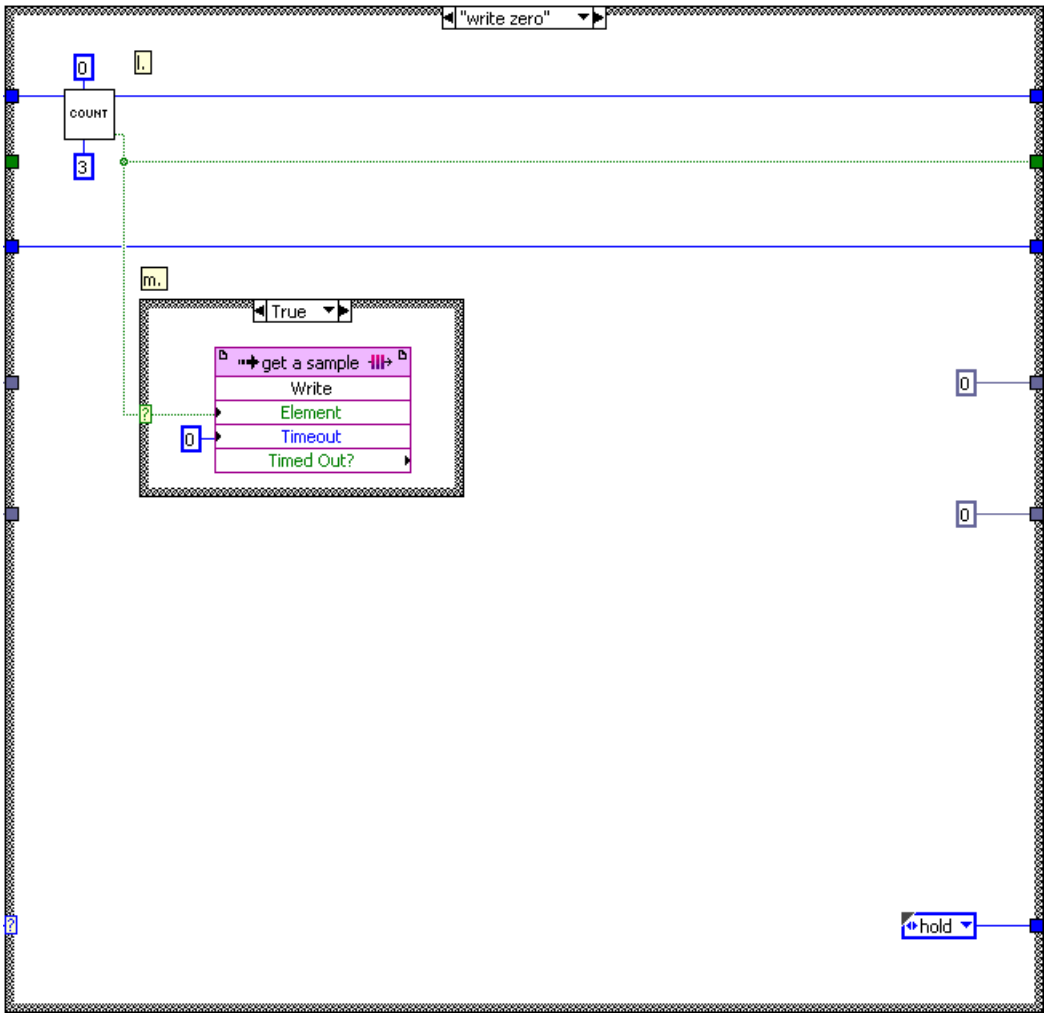
i1. e i2. *Memory Items* "symbol map re" e "symbol map im": Dados gravados em memória com os valores para os sinais I e Q.

j1. e j2. *Feedback Nodes*: Três *Feedback Nodes* conectados às memórias devido a latência das mesmas.

k. Constante numérica *enum*: Valor "hold" referente ao próximo estado da máquina de estados uma vez estando no estado "read sample".

E. *Input Node*: Recebe o valor do *clock* de referência que controla a frequência de execução do SCTL.

Figura 60 – Diagrama de blocos do caso “write zero” do *loop* de modulação

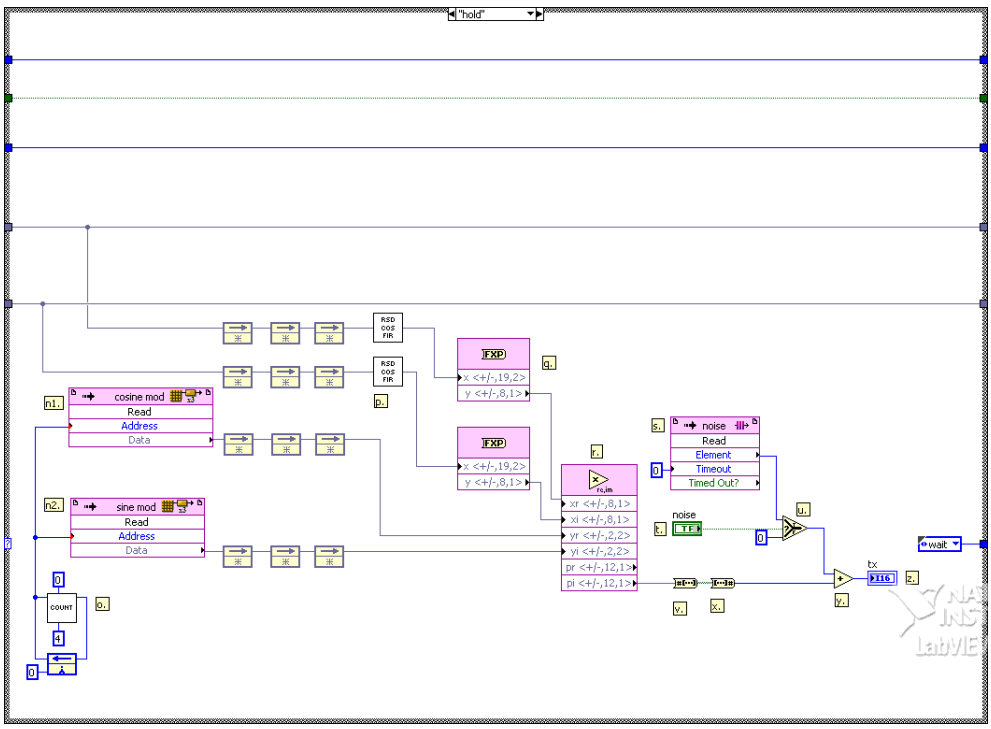


Fonte: Produzido pelo autor

l. Sub VI “counter”: VI utilizada em mais de uma parte do FPGA. Implementa a função de contador tendo os terminais de entrada e saída do contador, valor inicial, valor final e um terminal que sinaliza quando o contador chegou ao final (variável booleana).

m. *Case Structure*: Responsável pela escrita da variável *dummy* “get a sample” para a leitura de uma amostra do sinal analógico adquirido pelo conector de áudio analógico. É executado após o contador descrito em **i.** chegar ao final de sua contagem.

Figura 61 – Diagrama de blocos do caso "hold" do *loop* de modulação



Fonte: Produzido pelo autor

n1. e n2. *Memory Items* "cosine mod" e "sine mod": Dados numéricos para os valores que constituem uma onda cosenoidal e senoidal, ambas de expressas em 4 pontos, respectivamente. Constituem o sinal da portadora.

o. Sub VI "counter": contador sem a utilização de um *shift register* que controla a seleção dos dados lidos em **n1.** e **n2.**.

p. Sub VI "raised cosine filter": Contém a implementação do filtro tipo FIR *Raised Cosine*.

q. *High Throughput To Fixed Point*: faz a conversão de um dado do tipo ponto fixo a outro em diferentes configurações para o tamanho da parte inteira e de palavra.

r. *High Throughput Complex Multiply*: Realiza a multiplicação entre dois números complexos.

s FIFO "noise"': Recebe os valores numéricos, do RT, de um sinal do tipo ruído branco para ser somado ao valor escrito na saída do conector analógico MSP (AO0).

t. Controle Booleano: Controlado pelo RT. Em caso positivo, o ruído é somado ao sinal transmitido.

u. *Select*: Estrutura de seleção. Em caso positivo, soma-se ao sinal transmitido o ruído branco recebido pelo RT. Em caso negativo, soma-se ao sinal transmitido a constante numérica 0, i.e., não altera o sinal original.

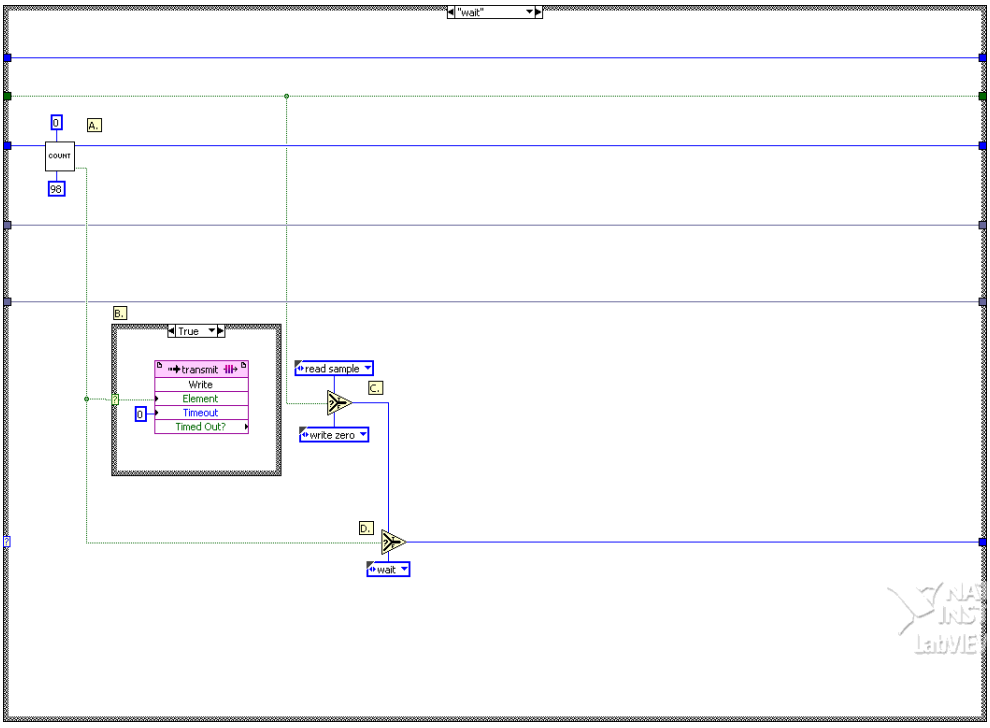
v. *Number To Boolean Array*: Converte um número em formato de ponto fixo para um *array* de valores booleanos.

x. *Boolean Array To Number*: Converte um *array* de valores booleanos para um valor numérico pré-configurado.

y. *Add*: Soma-se o valor numérico do sinal a ser transmitido ou com o ruído branco ou coma constante numérica 0.

z. Indicador numérico "tx": Variável que acumula o valor a ser transmitido. Será lido pelo *loop* de transmissão "tx".

Figura 62 – Diagrama de blocos do caso "wait" do *loop* de modulação



Fonte: Produzido pelo autor

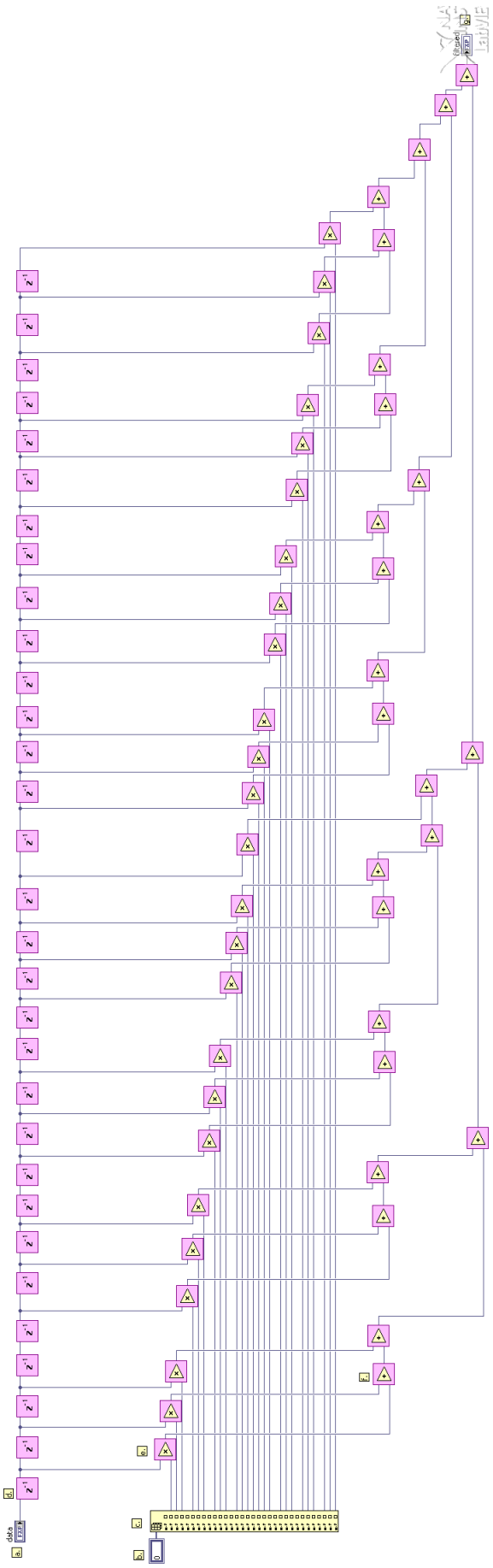
A. Sub VI "counter": Controla o número de vezes que a máquina de estados permanece no estado "wait".

B. *Case Structure*: É executada após o contador ter atingido o seu valor final. Em seu caso verdadeiro, a variável *dummy* booleana "transmit" é escrita. Esta controlará a frequência do sinal de transmissão feita no *loop* "tx".

C. *Select*: Seleciona o próxima estado da máquinas que será executado após o contador ter atingido o seu valor final. É controlado pela quantidade de número de zeros escritos (*oversampling*).

D. *Select*: Seleciona o próxima estado a ser executado pela máquina de estados. É controlado pelo contador descrito em **A.**. Em caso positivo seleciona um dos estados do *Select* descrito em **C.**, em caso negativo, retorna ao mesmo estado, "wait".

Figura 63 – Diagrama de blocos da sub VI “raised cosine filter fir”



Fonte: Produzido pelo autor

a. Controle númerico “data”: Valor de entrada para a VI. É o valor que será filtrado.

b. *Array Constant*: *Array* com os valores constante do coeficiente do filtro.

c. *Index Array*: Seleciona cada elemento do *array* dos coeficientes, do primeiro ao último. Elementos sem um “fio” conectado à saída são 0.

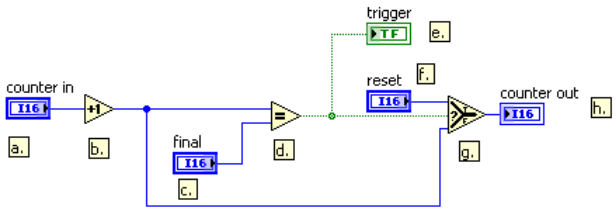
d. *Discrete Delay*: Acumula o valor entre uma iteração e outra. Possui a mesma função do *feedback node*.

e. *High Throughput Multiply*: Realiza a multiplicação entre dois números em aritmética de ponto fixo.

f *High Throughput Add*: Realiza a soma entre dois números em aritmética de ponto fixo.

g. Indicador numérico “filtered”: Recebe o resultado do valor filtrado de entrada “data”.

Figura 64 – Diagrama de blocos da sub VI “counter”



Fonte: Produzido pelo autor

a. Controle numérico “counter in”: Recebe o valor do contador a cada iteração.

b. *Increment*: Incrementa o número recebido em uma unidade.

c. Controle numérico “final”: Recebe o valor final do contador.

d. *Equal?*: Compara dois valores de entrada. Caso os dois valores sejam iguais escreve em sua saída o valor booleano falso, “T”, caso contrário, “F”.

e. Indicador booleano “trigger”: Exibe o valor recebido por “Equal?”, descrito no item **d.**.

f. Controle numérico “reset”: Recebe o valor de início para o contador. Este valor será selecionado quando o contador atingir o seu valor final.

g. *Select*: Seleciona entre o valor inicial e o corrente do contador. É controlado pelo resultado de “Equal?”, descrito no **item d.**.

h. Indicador numérico "counter out": Exige o valor do contador após o seu incremento e/ou valor de *reset*.

4.1.3.1 Objetivo

Modulação dos sinais coletados pelo *loop* de aquisição. Através leitura dos valores do sinal amostrado no *loop* de aquisição, faz-se a seleção, mapeamento, do valor inteiro lido para os respectivos valores I e Q. Em seguida procede-se para a modulação destes sinais. Após ter-se o valor da modulação, através de uma multiplicação complexa, seleciona-se a parte real para a transmissão, podendo ser somado ou não um sinal de ruído branco proveniente do RT.

4.1.3.2 Fluxo de dados

Dados de entrada:

- Variável local "signal", variável do tipo I8;
- Controle booleano "noise".

Dados de saída:

- Indicador numérico "tx", variável do tipo I16.

4.1.3.3 Otimizações realizadas

Criação das sub VI's "counter" e "raised cosine filter fir" para uma melhor estruturação do código. Utilização das funções *High Throughput Math* para uma melhor performance de cálculos. Utilizou-se a latência máxima para os *memory items* para que fossem implementados no *clock* de 10 MHz e para não utilizarem tantos recursos do FPGA. Utilizou-se uma arquitetura de máquina de estados para obter um controle otimizado para as frequências de aquisição e transmissão.

4.1.3.4 Tempo de execução

Como o *clock* de referência para este loop é de 10 MHz, cada iteração corresponde a 4 *ticks* de um clock de 40 MHz. Sendo 1 *tick* equivalente a 25 ns, o tempo de execução de cada ciclo de *clock* é de 100 ns.

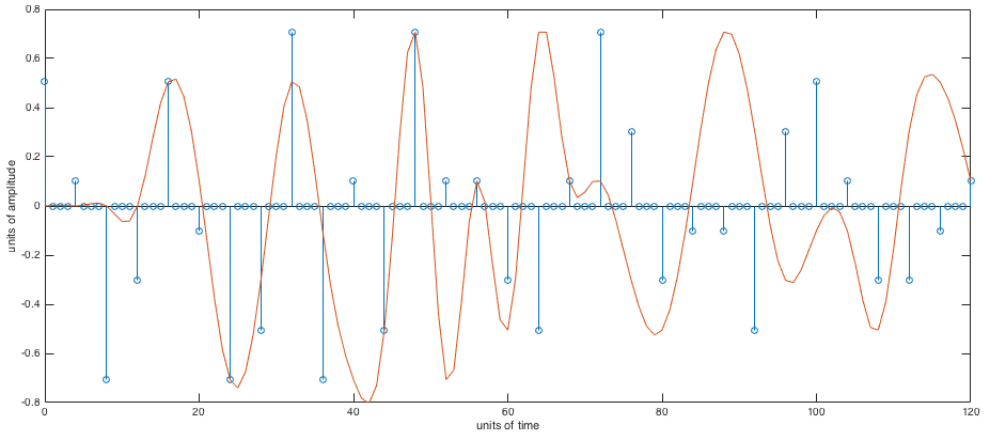
4.1.3.5 Descrição do funcionamento

Este *loop* possui uma máquina de estados com 4 estados distintos. Inicializa-se no estado "read sample". Este estado é responsável por ler a variável local "signal", proveniente do *loop* de aquisição do sinal de áudio. Ao valor de "signal" é somada a constante numérica

"31", variável do tipo I8, para que todos os valores recebidos pela entrada analógica estejam num intervalo de 0 a 63. Para que isto ocorra é necessário que a amplitude do sinal de entrada não ultrapasse os limites de -32 a +31, variável do tipo I16 (entrada de áudio analógica). Optou-se por esta limitação para que os valores inteiros recebidos pela entrada sejam diretamente mapeados aos valores do diagrama de constelação. Assim, o valor lido na entrada será diretamente remetido à posição equivalente ao seu valor. Desta maneira, não se transmite os valores *bit* a *bit* (0 ou 1) do sinal coletado, mas sim um conjunto de *bits* diferentes de 2. Esta é uma possibilidade que surge pelo fato do LabVIEW fornecer diretamente um número inteiro de 16 *bits*, mas que será convertido em uma variável de 8 *bits*, proveniente de sua saída analógica. Como há uma possibilidade de 64 valores para o sinal de entrada amostrado, 0 a 63, utilizou-se a modulação 64 QAM. Assim se tem uma resolução de 64 níveis distintos para o sinal de entrada. Os valores são mapeados através dos *memory items*, que são configurados para um número de ponto fixo na configuração <+-16,1>. Os valores correspondentes, valores dos sinais I e Q, são colocados nos *shift registers* **e.** e **d.**, Figura 61, para leitura no estado seguinte.

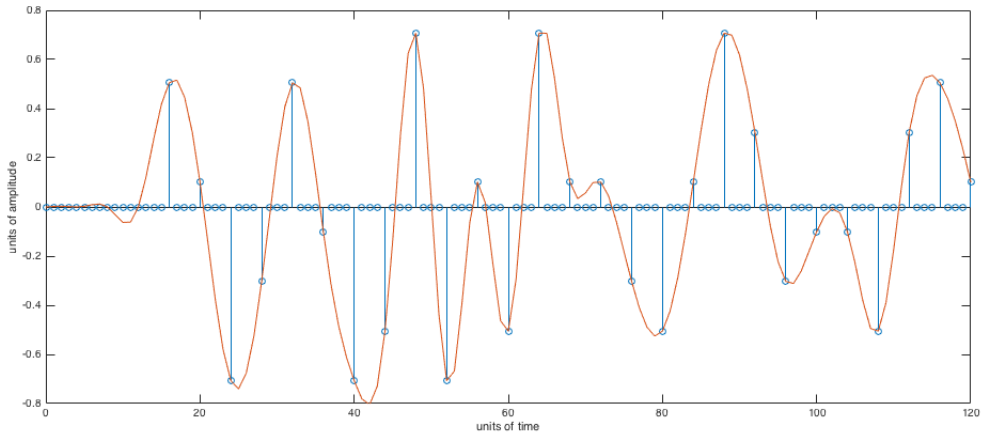
No estado "hold" os sinais I e Q são filtrados através da sub VI "raised cosine fir" com o intuito de minimizar a interferência intersimbólica (ISI). Como se pode ver na sub VI "raised cosine fir", implementou-se o filtro através de somas e multiplicações a equação deste filtro. As funções *High Throughput Math* são utilizadas para otimizar as operações. Os valores dos coeficientes do filtro são armazenados em um *array* constante e selecionados um a um para cada operação das somas e multiplicações a que lhes convém. Utilizando-se o bloco *Delay*, que possui a mesma funcionalidade que um *feedback node*, os valores entre iterações são armazenados. Para uma melhor visualização do *delay* inserido pelo filtro, a Figura 65 mostra o sinal de entrada, em azul, e o sinal de saída, em vermelho. Deslocando-se o gráfico da entrada do filtro em 16 unidades, *delay* causado pelo filtro, verifica-se a suavização do sinal de saída, que minimizam as transições entre valores apresentado na Figura 66.

Figura 65 – Sinal de entrada (azul) do filtro e seu sinal de saída (vermelho)



Fonte: Produzido pelo autor

Figura 66 – Sinal de entrada (azul) e sinal de saída (vermelho)do filtro com o deslocamento do sinal de entrada



Fonte: Produzido pelo autor

Uma vez estando os valores filtrados é feita uma multiplicação complexa entre estes e o sinal da portadora, proveniente das *memory items* “cosine mod” e “sine mod”. O controle dos valores a serem escritos por elas são feitos pela sub VI “counter”, que possui um período de 4 iterações, ou seja, passagens pelo estado “hold”. De acordo com a Figura 64, esta sub VI recebe o sinal atual do *shift register* e o incrementa em uma unidade. Em seguida é verificado se o valor atual é igual ao valor de *reset*, caso seja, a estrutura *Select* seleciona o valor inicial para o contador reiniciando-o. Em caso negativo, o valor atual é

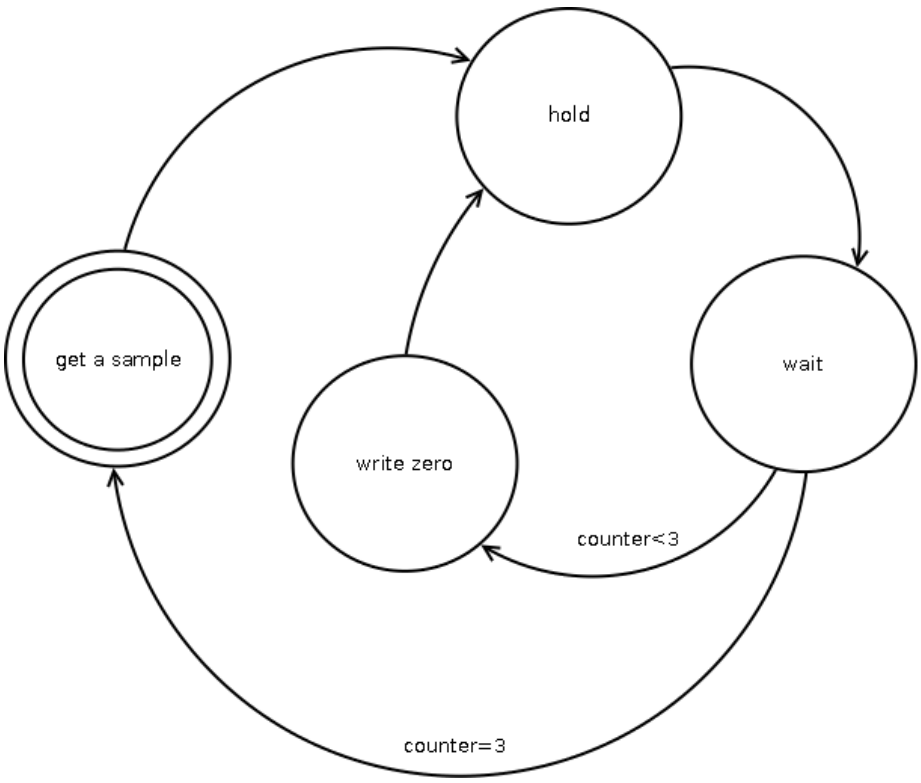
passado ao *shift register* para a próxima iteração. Como a latência dos valores da portadora é de três ciclos de *clock*, são necessários três *feedback nodes* não inicializados conectados às saídas destas. Como os valores dos sinais I e Q precisam estar sincronizados com os resultados válidos do sinal da portadora, também recebem três *feedback nodes* antes de sua filtragem. O resultado da multiplicação complexa não possui latência e fica disponível em um ciclo de relógio. O escrito, que será transmitido, no indicador numérico "tx", que também é uma variável local para o *loop* de transmissão, "tx" deve ser convertido em um valor inteiro do tipo I16. Para que isto seja feito, é utilizado os conversores "Number To Boolean Array" seguido de "Boolean Array To Number". Desta maneira a conversão ocorre sem erros e perda de informação (IP..., 2009). Dependendo do controle booleano "noise", comandado pelo RT, é possível somar-se ao sinal transmitido ruído do tipo branco através da FIFO "noise", feito pelo item *Select*.

A seguir, no estado "wait", a sub VI "counter" controla um número de 98 iterações neste estado. Ao se atingir o valor final deste contador, a *case structure* escreve o valor da variável *dummy* na FIFO "transmit" para que o sinal do indicador numérico "tx" seja lido pelo *loop* de transmissão e escrita na saída analógica do myRIO. O próximo estado dependerá do número de zeros inseridos entre as amostras colhidas do sinal de entrada. Caso não tenham sido escritas 3 amostras de valor 0, o *oversampling*, após cada amostra do sinal de entrada, o próximo estado será "write zero", caso contrário, "get a sample".

No estado "write zero" os valores numéricos constantes 0 são colocados nos *shift registers* dos sinais I e Q; a sub VI "counter" incrementa em uma unidade cada passagem por este estado. Estando neste estado, o próximo sempre será "hold".

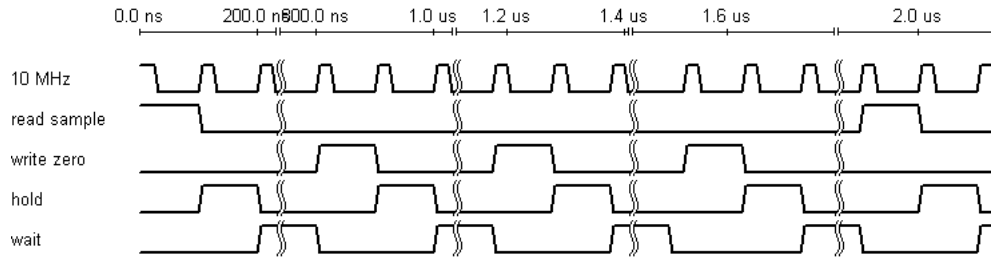
De maneira geral o ciclo da máquina de estados sempre seguirá a mesma sequência de acordo com o diagrama de estados da Figura 67.

Figura 67 – Máquina de estados seguida pelo laço de modulação



Fonte: Produzido pelo autor

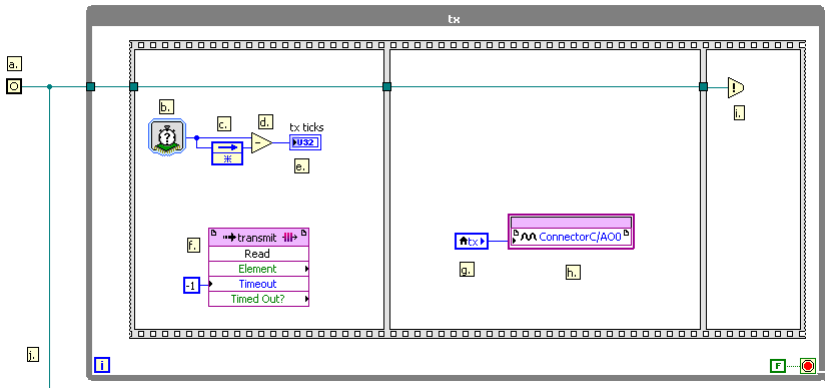
Figura 68 – Diagrama temporal da máquina de estados presente no *loop* "modulation loop"



Fonte: Produzido pelo autor

4.1.4 Transmissão do sinal modulado - *tx loop*

Figura 69 – Diagrama de blocos do *loop* tx



Fonte: Produzido pelo autor

a. *Occurrence* Controla o sincronismo entre os *loops* de transmissão "tx" e o de recepção "rx".

b. Sub VI "Tick Count": Contagem das ocorrências de transmissão.

c. *Feedback Node*: Armazena o valor anterior em relação ao atual para calcular a diferença de *ticks* entre duas iterações.

d. *Subtract*: Realiza a subtração entre os *ticks* anterior e atual do *loop*.

e. Indicador numérico "tx ticks": Exibe o resultado da diferença dos *ticks* mencionados no item **d.**.

f. FIFO "transmit": Faz a leitura do sinal *dummy* escrito no *loop* de modulação. Controla o perído de escrita do sinal analógico de saída.

g. Variável numérica local "tx": Faz a leitura do sinal a ser transmitido.

h. Conector de saída analógico AO0: Escreve o sinal "tx" na saída analógica.

i. *Set Occurrence*: Sinaliza que a ocorrência ocorreu. Esta ocorrência será esperada pelo *loop* de recepção "rx".

j. "Fio" de ligação entre a ocorrência e sua espera.

4.1.4.1 Objetivo

Escrita da variável numérica "tx" no conector analógico da saída AO0. Sincronismo entre o conector de saída e o de entrada AI0 através de uma ocorrência.

4.1.4.2 Fluxo de dados

Dados de entrada:

- FIFO "transmit", *dummy signal* booleano (verdadeiro ou falso);
- Controle numérico "tx", variável do tipo I16.

Dados de saída:

- Indicador numérico "tx ticks", variável do tipo U32.
- Conector AO0, variável do tipo I16;
- *Set Occurrence*, variável do tipo *occurrence*.

4.1.4.3 Tempo de execução

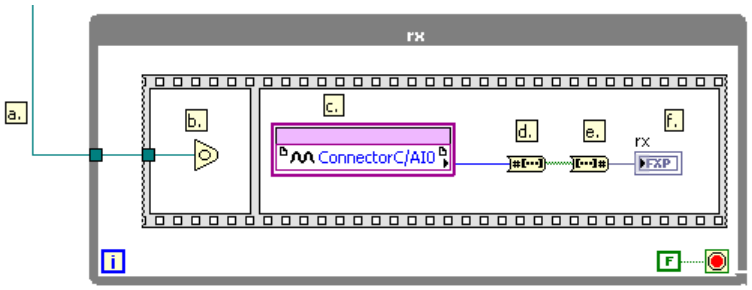
Controlada pela variável *dummy* booleana através da leitura da FIFO "transmit", que é escrita a cada 400 *ticks* de um *clock* de referência de 40 MHz, ou seja, 10 µs ou 100 kHz.

4.1.4.4 Descrição do funcionamento

Este *loop* só é executado após a variável *dummy* da FIFO "transmit" ser escrita, pois a sua leitura é controlada por um *timeout* configurado em -1, ou seja, de espera indevida, o que significa que a FIFO ficará em estado *idle* até que um novo valor esteja disponível para a leitura. Quando houver um sinal disponível, é lido o valor da variável local "tx", escrita no estado "hold" do *loop* de modulação e escrito na saída analógica AO0, localizado no segundo *frame* da *stack sequence*. No terceiro, é dado o sinal de ocorrência para o *loop* de recepção, ‘"rx".

4.1.5 Recepção do sinal modulado - *rx loop*

Figura 70 – Diagrama de blocos do *loop* rx



Fonte: Produzido pelo autor

a. "Fio" recebido da ocorrência controlada pelo *loop* tx.

b. *Wait Occurrence*: somente executado quando a ocorrência é escrita, *Set Occurrence*, no *loop* tx.

c. Conector Analógico AI0 de entrada: Recebe os valores analógicos do sinal modulado transmitidos por AO0.

d. *Number To Boolean Array*: Converte um número I16 para um *array* de valores booleanos.

e. *Boolean Array To Number*: Converte um *array* de valores booleanos para um valor numérico pré-configurado, neste caso <+-12,1>.

f. Indicador numérico de ponto fixo: variável local para o *loop* de demodulação.

4.1.5.1 Objetivo

Realiza a leitura dos valores recebidos no conector analógico de entrada e a conversão deste valores, I16, em um número ponto fixo <+-12,1> para ser lido, como variável local, pelo *loop* de demodulação.

4.1.5.2 Fluxo de dados

Dados de entrada:

- *Wait On Occurrence*, variável do tipo *occurrence*.
- Conector AI0, variável do tipo I16;

Dados de saída:

- Indicador numérico "rx", variável do tipo ponto fixo <+-12,1>.

4.1.5.3 Tempo de execução

Como cada iteração necessita da ocorrência controlada pelo *loop* de transmissão, possui a mesma frequência de operação do *loop* tx, 100 kHz ou 10 µs.

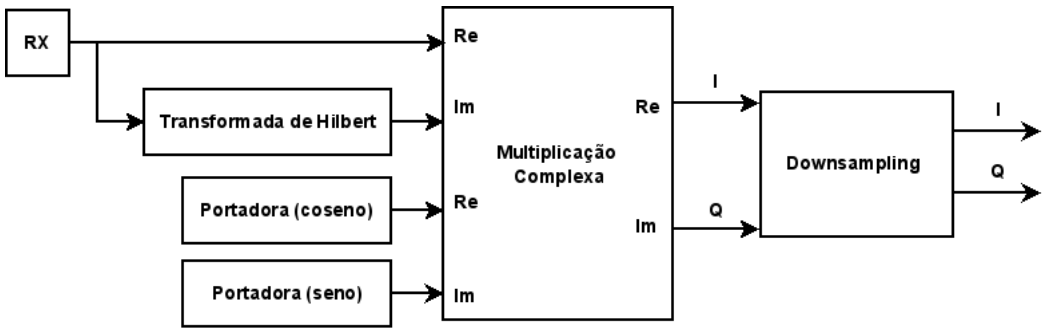
4.1.5.4 Descrição do funcionamento

Ao receber o sinal da ocorrência dado pelo *loop* de transmissão, executa a leitura da entrada analógica. O valor recebido, em I16, é convertido para ponto fixo e escrito na variável local "rx", que será lida pelo *loop* de demodulação.

4.1.6 Demodulação do sinal - *demodulation loop*

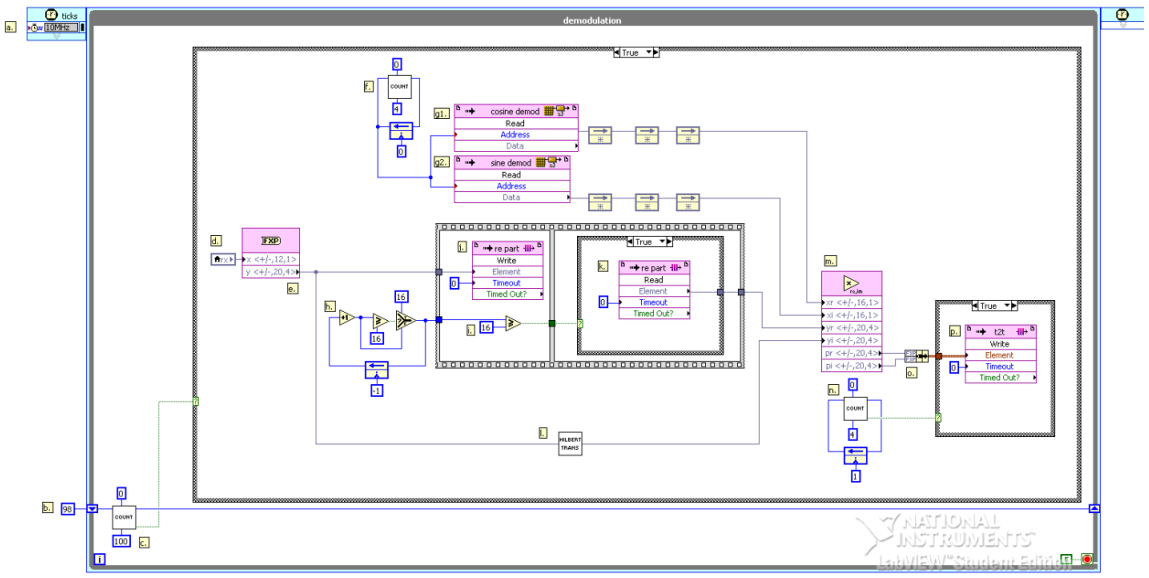
A demodulação do sinal ocorre de maneira semelhante à modulação, utilizando-se a multiplicação complexa entre o sinal recebido e o sinal da portadora. O sinal recebido será a parte real e esse sinal, após ser processado pelo filtro que implementa a transformada de Hilbert, recuperará a sua parte imaginária. Após o resultado da multiplicação ser obtido é feito o *downsampling* do sinal (retirada dos zeros) e assim se recupera o sinal originalmente transmitido. O diagrama da Figura 71 descreve o processo.

Figura 71 – Demodulação do sinal



Fonte: Produzido pelo autor

Figura 72 – Diagrama de blocos do *loop demodulation*



Fonte: Produzido pelo autor

a. *Input Node*: Recebe o valor do *clock* de referência que controla a frequência de execução do SCTL.

b. Constante numérica: Inicia o *shift register* com o valor 98, contante do tipo I16. *Shift register* responsável pelo controle de leitura do dado "rx", através da *case structure*, do *loop* de recepção.

c. Sub VI "Counter": Controla em qual iteração a *case structure* será executada.

d. Variável local "rx": Leitura da variável local escrita no *loop* de recepção.

e. *High Throughput To Fixed Point*: faz a conversão de um dado do tipo ponto fixo a outro em diferentes configurações para o tamanho da parte inteira e de palavra.

f. Sub VI "counter": contador sem a utilização de um *shift register* que controla a seleção dos dados lidos em **g1.** e **g2.**.

g1. e g2. *Memory Items* "cosine demod" e "sine demod": Vetor de dados numéricos previamente armazenados em memória para os valores que constituem uma onda cosenoidal e senoidal, ambas expressas em 4 pontos por ciclo, respectivamente. Constituem o sinal da portadora.

h. Contador: Estrutura que controla a leitura da FIFO *target-scope* "re part".

i. *Greater Or Equal?*: Compara se o valor do contador descrito em **h.** é maior ou igual à constante numérica, I16, de valor 16.

j. *Write* FIFO "re part": FIFO que escreve os valores recebidos do sinal recebido "rx".

k. *Read* FIFO "repart": Realiza a leitura dos valores da FIFO se o contador que a controla é igual ou superior ao valor de 16.

l. Sub VI "hilbert fir": Implementação da transformada de Hilbert, sob a forma de um filtro do tip FIR.

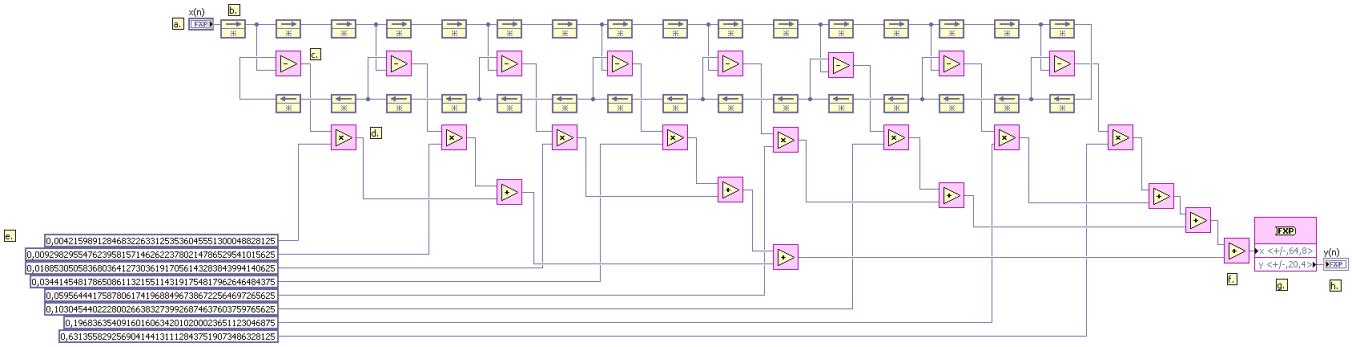
m. *High Throughput Complex Multiply*: Realiza a multiplicação entre dois números complexos.

n. Sub VI "counter": Contador que controla quais amostras do sinal demodulado serão escritas na FIFO "t2t".

o. *Bundle*: Cria um *cluster* com os sinais I e Q demodulados.

p. *Write* FIFO "t2t": FIFO do tipo *target-scope* que repassa os valores, I e Q, para o *loop* externo "to host".

Figura 73 – Diagrama de blocos da sub VI "hilbert fir"



Fonte: Produzido pelo autor

a. Controle numérico "x(n)": Recebe os valores a serem aplicados à transformada de Hilbert, variável do tipo ponto fixo <+-20,4>.

b. *Feedback Node*: Acumula os valores numéricos entre duas iterações.

c. *High Throughput Subtract*: Faz a subtração entre dois números.

d. *High Throughput Multiply*: Realiza a multiplicação entre dois números.

e. Constantes numéricas: Representam os coeficientes do filtro FIR para a implementação da transformada de Hilbert.

f. *High Throughput Add*: Faz a adição entre dois números.

g. *High Throughput To Fixed Point*: Faz a conversão de um dado do tipo ponto fixo a outro em diferentes configurações para o tamanho da parte inteira e de palavra.

h. Indicador numérico "y(n)": Exige os valores resultantes da filtragem do sinal de entrada x(n).

4.1.6.1 Objetivo

Ler os dados recebidos pelo conector analógico de entrada, a partir do *loop* "tx", realizar filtragem e demodulação. Fazer o *downsampling* e a transferência dos dados para o *loop* "to host".

4.1.6.2 Fluxo de dados

Dado de entrada:

- Indicador numérico "rx", variável do tipo ponto fixo <+-12,1>.

Dado de saída:

- FIFO "t2t", variável do tipo cluster de dois elementos numéricos do tipo ponto fixo <+-20,4>.

4.1.6.3 Otimizações realizadas

Para todas operações matemáticas de ponto fixo utilizou-se as funções *High Throughput Math*. Utilizou-se também a latência máxima das memórias, ligando-as com 3 *feedback nodes* em suas saídas. O processo de *downsampling* foi incluído neste loop, no entanto, via uma FIFO do tipo *target-scope*. Isto foi feito para que houvesse sincronismo entre os valores correspondentes dos sinais I e Q durante a transferência destes valores para o *host*. Assim, a transferência ocorre através de um *loop* for, escrevendo-se primeiro o valor I e em seguida o valor Q.

4.1.6.4 Tempo de execução

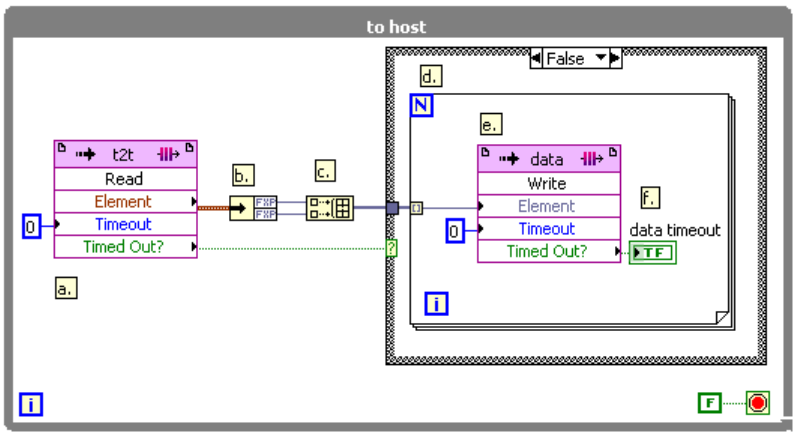
Seu tempo de execução está sincronizado com o *loop* de leitura da entrada analógica AI0. No entanto, como se encontra dentro de um SCTL, é utilizado um contador para fazer o controle de ciclos de *clock* nos quais devem ser lidos os valores do valor recebido. Assim, utilizou-se um valor de *clock* de referência de 10 MHz, o que representam uma iteração de 4 *ticks*, ou seja, 100 ns. Como o *loop* de recepção é executado a uma taxa de 400 *ticks*, para cada *tick* é necessário que o *loop* de demodulação seja executado uma vez, estabelecendo a relação 1 para 100.

4.1.6.5 Descrição do funcionamento

A partir do valor recebido pelo *loop* de leitura do conector analógico AI0, o dado é convertido a um valor de ponto fixo adequado para a VI "hilbert fir", que realiza a transformada de Hilbert. Como o *delay* causado pelo filtro resulta em um intervalo de tempo total de 16 amostragens, é utilizada uma FIFO, "re part", para armazenar os 16 primeiros valores recebidos. Para que isto ocorra um contador é implementado iniciando-se em -1, sendo incrementado até atingir o valor 16. Ao chegar neste limite, o contador permanece com este valor, através da estrutura de seleção *Select*. Estando neste valor, os valores escritos na FIFO começam a ser lidos. Além da FIFO, a VI "hilbert fir" também recebe os valores da transmissão, no entanto, diretamente, sem a necessidade de um *buffer*. Nesta VI, acontece o processo de recuperação da parte imaginária do sinal transmitido. Através da implementação desta transformada em um filtro do tipo FIR, tem-se o sinal gerado (parte imaginária), porém não transmitido, no *loop* de modulação. No entanto, nesta etapa este se torna necessário para o processo de demodulação. Seus coeficientes são simétricos e em um total de 8 coeficientes no formato de ponto fixo. A FIFO começa a ser lida e seu valor entregue ao bloco *High Throughput Complex Multiply* juntamente com os valores das ondas cosenoidais e senoidais da portadora, assim como a parte imaginária, sinal Q. Os valores da portadora encontram-se armazenados em memória e seu controle é feito através de um contador, idêntico ao processo realizado no *loop* de modulação. *Feedback Nodes* são conectados nas saídas das memória, pois devido a sua latência isto se torna obrigatório para o processo de compilação da VI. A partir da multiplicação complexa os sinais originais são recuperados e unidos através de um elemento *bundle*. Isto é feito para que se tenha o sincronismo entre os dois sinais, I e Q, para que permaneçam coerentes entre si. Estes são gravados na FIFO "t2t" e passados ao *loop* externo "to host" para correta transferência dos dados ao *host*.

4.1.7 Passagem de dados do FPGA para o RT - *to host loop*

Figura 74 – Diagrama de blocos do loop “to host”



Fonte: Produzido pelo autor

a. FIFO “t2t”: Leitura dos sinais I e Q provenientes do *loop* de demodulação.

b. Elemento *unbundle*: Separa os elementos de um *cluster*.

c. *Build Array*: Constrói um *array* de dois elementos.

d. *Loop for*: Escreve os elementos do *cluster* de maneira sequencial na FIFO “data”.

e. FIFO ”data”: Recebe os valores dos sinais I e Q para serem transferidos ao *host*.

f. Indicador booleano “data timeout”: Exige a ocorrência da FIFO ter sinalizado um estado de *timeout*, se não há espaço para novos elementos serem escritos.

4.1.7.1 Objetivos

Transferir os valores dos sinais I e Q para o *host* de uma maneira sequencial.

4.1.7.2 Fluxo de dados

Dado de entrada:

- *Read* FIFO “t2t”, variável do tipo *cluster* de dois elementos do tipo ponto fixo <+-20,4>.

Dados de saída:

- *Write* FIFO “data”, variável do tipo número inteiro do tipo ponto fixo <+-20,4>.

- Indicador booleano “data timeout”, variável do tipo booleana.

4.1.7.3 Tempo de execução

Este *loop* é executado a cada 9 *ticks* com um *clock* de referência de 40 MHz, onde 1 *tick* corresponde a 25 ns. Assim sendo, seu período é de 225 ns ou 4.44 MHz.

4.1.7.4 Descrição do funcionamento

Através da FIFO “t2t” os valores dos sinais I e Q são separados através de um elemento *unbundle* e estes são agrupados em forma de um *array* de dois elementos, onde o primeiro é o sinal I e o segundo o sinal Q. Estes elementos são passados a outra FIFO, “data”, para serem transferidos ao *host*. Porém, só serão escritos dados em “data” enquanto a FIFO “t2t” não sinalizar um sinal de *timeout*, que pode ocorrer caso ela não tenha dados novos a serem lidos, ou seja, esteja vazia. Para prevenir que isto ocorra, seu terminal de *timeout* é ligado a uma *case structure*, de maneira que somente quando o terminal possuir um valor negativo, isto é, existam valores a serem lidos, ou seja, dados válidos, eles possam serem gravados na FIFO “data”. Desta maneira, a FIFO “data” encontra-se no caso “False” dessa estrutura. Utiliza-se um *loop* for de maneira que o *array* de entrada, sinais I e Q, esteja sendo indexado, ou seja, cada iteração receberá um elemento do *array*. Como o *array* possui somente dois elementos, este *loop* só será executado duas vezes. Isto é feito para que os sinais sejam colocados de maneira alternada na FIFO elemento a elemento. Isto traz uma vantagem para o *host* no momento da leitura destes elementos, pois como os elementos são gravados um a um, no *host* serão lidos de grupos a grupos, isto é, de *arrays* em *arrays*. No entanto, do ponto de vista de entrada de dados, constituem um único *array*. Para se obter a coerência necessária destes dados, para interpretá-los corretamente, é necessário utilizar o bloco “Decimate Array” que realiza justamente a função de separar os dados escritos na FIFO, elemento a elemento, em dois *arrays* distintos, facilitando assim a leitura destes elementos, que virão sincronizados entre si mas ao mesmo tempo, separados. No terminal de *timeout* da FIFO *Read* “data” é colocado um indicador booleano para sinalizar se houve ou não *timeout* para esta FIFO, caso o *host* não “esvazie” os dados contidos a tempo de novos dados serem escritos. Este indicador é lido no *host* e caso exiba valores verdadeiros, indicando o *timeout*, é necessário aumentar a frequência de leitura e/ou o número do bloco de elementos lidos.

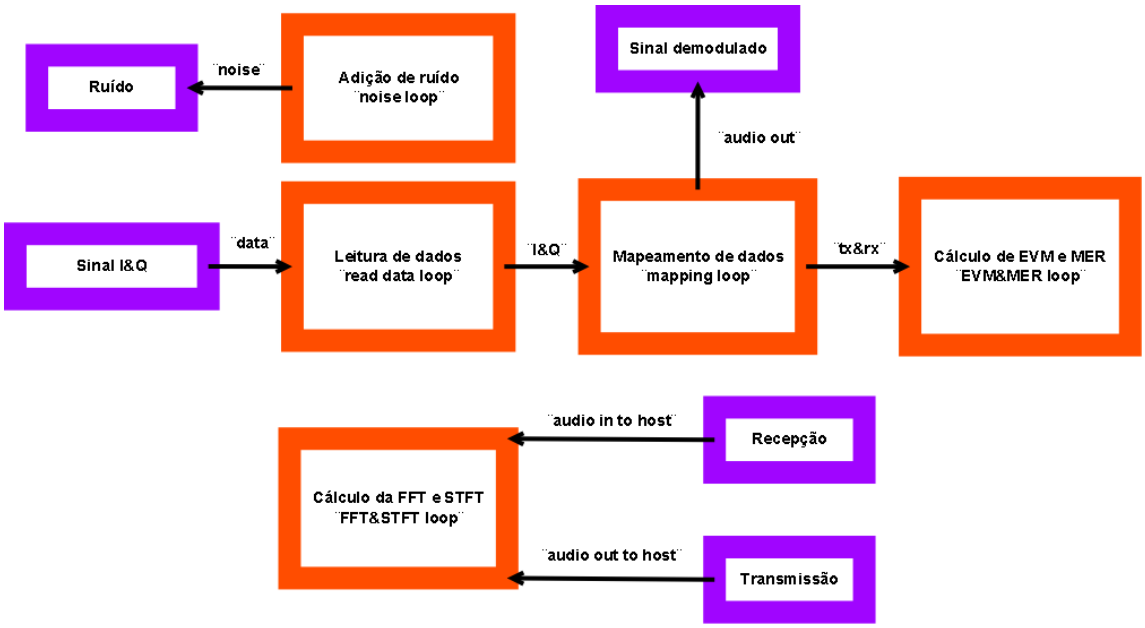
4.2 RT

Os dados obtidos no FPGA são transferidos ao RT, através de FIFO’s, para serem analisados. Constituem um conjunto de três dados distintos: o sinal recebido do conector de entrada de áudio, o sinal escrito na saída de áudio e o sinal demodulado. Os dois

primeiros são utilizados para visualização dos dados através de transformada rápida de Fourier, para a exposição do espectro de frequências do sinal adquirido e recuperado, e da transformada de termo curto de Fourier, para analisar a variação do espectro de frequência ao longo do tempo. O sinal demodulado é recebido e mapeado de acordo com o mapa de constelação da modulação 64 QAM, sendo enviado novamente ao FPGA para ser escrito na saída de áudio.

Há 5 processos ocorrendo em paralelo no RT: envio de ruído ao sinal transmitido, leitura do sinal demodulado no FPGA, mapeamento destes dados, cálculo do EVM e MER e cálculo da FFT e STFT. O primeiro e o último processos são independentes entre si e dos demais. Já os outros três estão interconectados, pois dependem do fluxo de dados do *loop* de recebimento do sinal demodulado. Logo, trabalham sincronizados entre si, onde o *loop* de controle é o "read data loop". Na Figura 75 há um fluxograma dos processos para uma melhor visualização do fluxo de dados entre o FPGA e o RT.

Figura 75 – Fluxo de dados no RT. Na cor roxa, os dados trocados com o FPGA através das FIFO's indicadas. Na cor vermelha, somente processos que ocorrem no RT

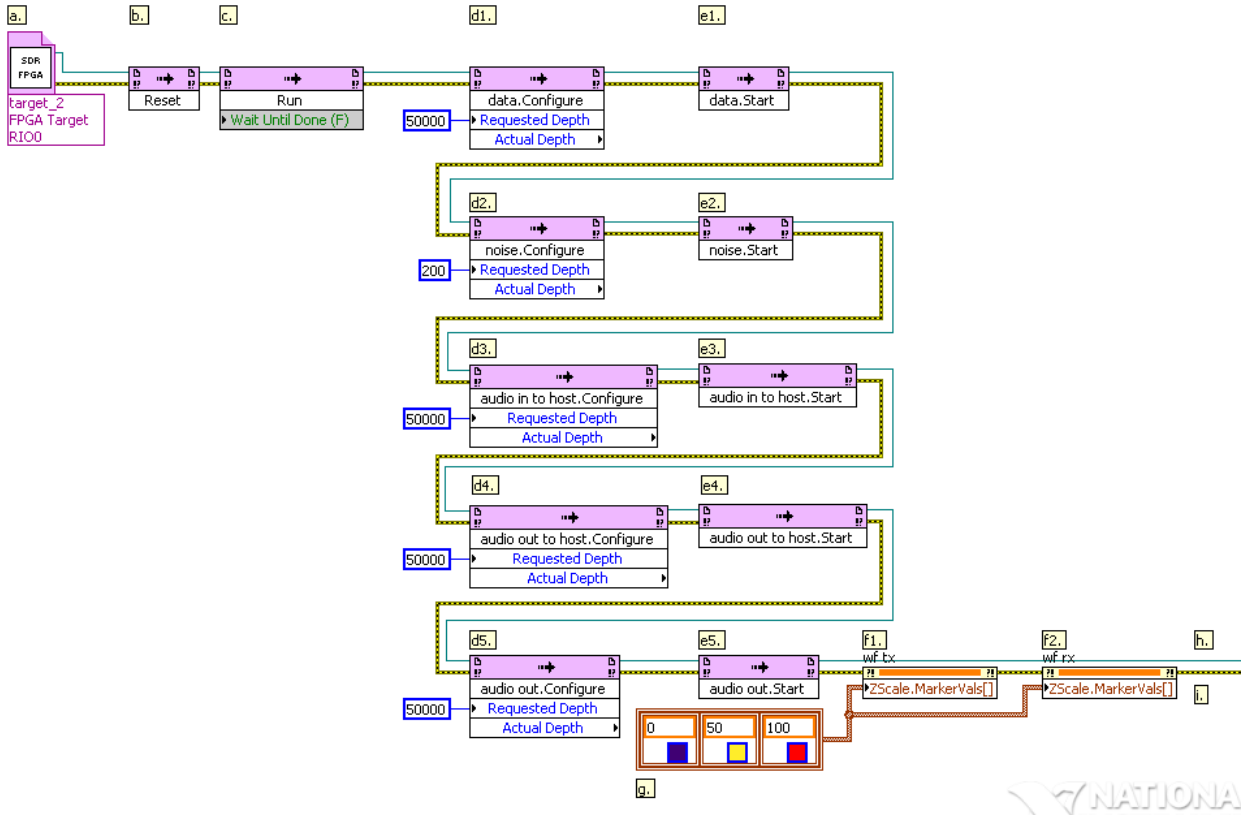


Fonte: Produzido pelo autor

Para um melhor entendimento de como o processamento de dados ocorre no RT, serão analisados primeiramente as configurações das FIFO's entre FPGA e RT e em seguida a função operante de cada *loop*.

4.2.1 Configuração inicial

Figura 76 – Diagrama de blocos das configurações dos itens do tipo FIFO, *Invoke Method* e *Property Node*



Fonte: Produzido pelo autor

a. *Open FPGA VI Reference*: Seleciona a VI no FPGA que será executada no target, assim como traz a informação de todos itens que podem ser configurados no RT.

b. *Invoke Method Reset*: Coloca em modo *reset* a VI presente no *target*.

c. *Invoke Method Run*: Inicializa a VI presente no *target*.

d1. a d5. Configura o número de elementos para o *buffer* presente no RT para as FIFO's do tipo *target to host* e *host to target*.

e1. a e5. Inicializa as FIFO's.

f1. e f2. *Property Nodes Marker Values*: Configura a escala do eixo z do gráfico do tipo *Intensity Chart*.

g. Constante do tipo *cluster*: Configurações para as escalas dos eixos z's dos gráficos "wf tx" e "wf rx".

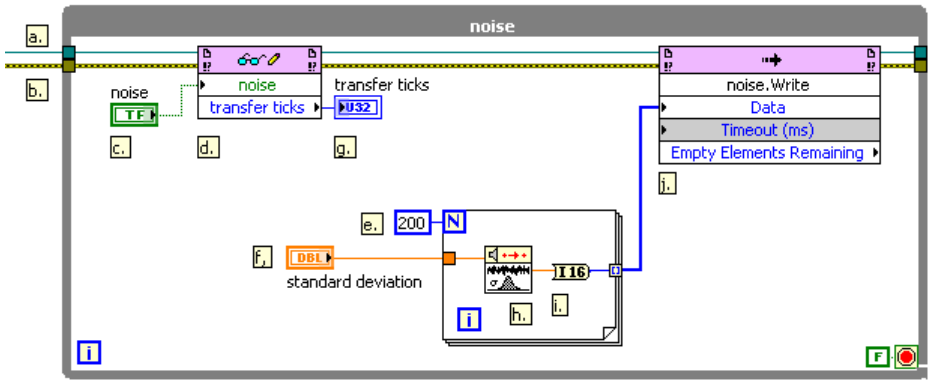
h. "Fio" da referência da VI utilizada no FPGA.

i. "Fio" dos dados de erro: *Cluster* de três elementos, *status*, *code* e *source*.

Inicialmente é aberta a referência da VI que se encontra no FPGA. A primeira ação é de se resetar a VI, para que esteja com as configurações iniciais para o seu início. Isto se faz necessário caso a utilização prévia desta VI não tenha sido encerrada de maneira correta, isto é, o esvaziamento dos elementos das FIFO's pode não ter sido feito. Assim, há a garantia de que os *buffers* estarão vazios. Em seguida a VI é colocada em operação. Todas as FIFO's utilizadas são previamente configuradas antes dos dados serem lidos por qualquer *loop* presente no *host*. Assim, configura-se o tamanho do *buffer* presente no RT através da estrutura *Invoke Method Configure*; dá-se início aos canais DMA através da estrutura *Invoke Method Start*. Também configuram-se as escalas dos eixos z dos gráficos *Intensity Chart* para os sinais recebidos e recuperados. Estes gráficos serão utilizados para a visualização do diagrama de cascata (*waterfall*). Os dados da referência da VI presente no *target* e o "fio" que contém os possíveis erros ocorridos durante a execução de algum item anterior são passados a diante para o próximo nó.

4.2.1.1 Geração de ruído branco - *noise loop*

Figura 77 – Diagrama de blocos do *loop noise*



Fonte: Produzido pelo autor

a. "Fio" da referência da VI utilizada no FPGA.

b. "Fio" dos dados de erro: *Cluster* de três elementos, *status*, *code* e *source*.

c. Controle booleano "noise": Variável do tipo booleana que seleciona a adição de ruído do tipo Gaussiano ao sinal transmitido.

d. *Read/Write Control*: Lê valores dos indicadores presentes no FPGA ou escreve valores de controles presentes no FPGA. Neste caso, escreve valores na variável booleana "noise" e lê valores do indicador numérico, U32 (*unsigned char* de 32 bits), "transfer ticks".

e. Constante numérica: Valor fixo de 200, correspondente ao número de iterações do laço *for*.

f. Controle numérico do tipo *double*: Determina o nível de ruído gaussiano inserido no sinal de transmissão.

g. Indicador numérico do tipo U32: apresenta os valores de "transfer ticks" presente no FPGA, localizado no *loop* de transferência de dados do *target* para o *host*.

h. Sub VI *Gaussian White Noise PtByPt*: VI que gera, ponto a ponto, valores de um sinal de ruído do tipo gaussiano.

i. Conversor numérico: Converte um número do tipo *double* para um número inteiro de 16 *bits*.

j. *Invoke Method Write*: Envia um *array* de dados para o FPGA.

4.2.1.2 Objetivos

Este *loop* tem a função de enviar ao *loop* de modulação, embarcado no FPGA, o sinal de um ruído do tipo gaussiano.

4.2.1.3 Fluxo de dados

Dados de entrada:

- Controle booleano "noise", variável do tipo booleana;
- Controle numérico "standard deviation", variável do tipo *double*, 64 *bits*, 15 dígitos de precisão;

Dados de saída:

- Indicador numérico "transfer ticks", variável do tipo U32;
- FIFO *Write* "noise" variável do tipo I16.

4.2.1.4 Otimizações realizadas

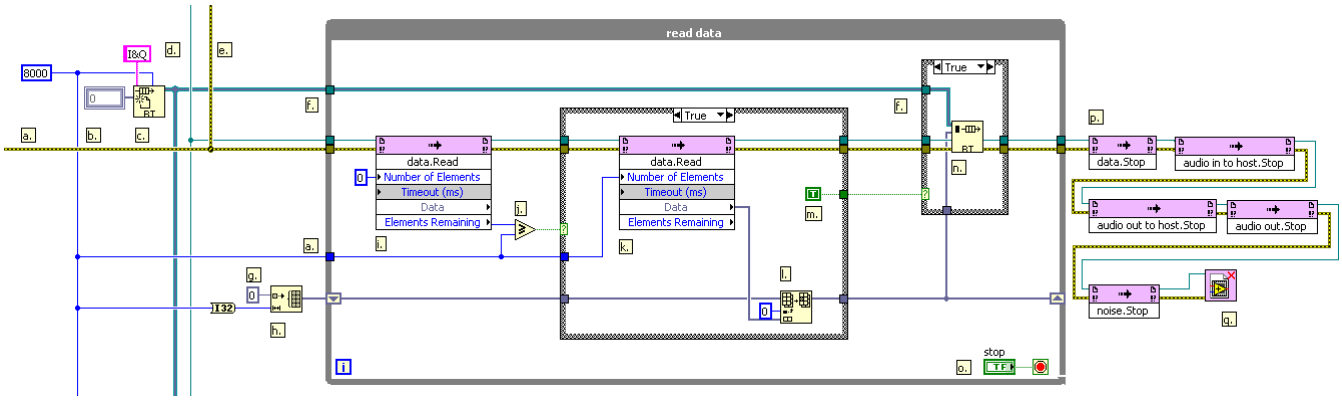
Utilizou-se a conversão do número do tipo *double* para I16 dentro do *loop for*, otimizando assim a utilização da memória de programa.

4.2.1.5 Descrição do funcionamento

Esta VI envia dados ao FPGA a cada iteração. O conjunto dos valores para o sinal do ruído gaussiano é gerado a partir da sub VI "Gaussian White Noise PtByPt" interna a um *loop* for que é executado por 200 iterações, gerando assim um *array* de 200 elementos. A amplitude do sinal é configurada através do controle numérico "standard deviation" e pode assumir valores de 0 a 500, com um incremento de 5. Estes valores só são adicionados ao sinal transmitido se o controle booleano "noise" possuir valor positivo. Isto é feito para que se possa controlar quando o ruído será inserido. Para que os dados sejam transferidos ao FPGA, faz-se a transferência destes dados através da FIFO "noise".

4.2.2 Recebimento de dados do FPGA - *read data loop*

Figura 78 – Diagrama de blocos do *read data loop*



Fonte: Produzido pelo autor

a. Constante numérica: Configura a quantidade de elementos do *array* de dados armazenado na RT FIFO "I&Q" para transferência.

b. *Array* de número do tipo ponto fixo: Configura o tipo de dados que serão recebidos pela RT FIFO "I&Q".

c. RT FIFO *Create*: Cria uma FIFO do tipo RT. Seus terminais são conectados aos itens descritos em **a.** e **b.**.

d. "Fio" da referência da VI utilizada no FPGA.

e. "Fio" dos dados de erro: *Cluster* de três elementos, *status*, *code* e *source*.

f. "Fio" de conexão para a RT FIFO "I&Q".

g. Constante numérica que configura o tipo de variável da estrutura *Initialize Array*.

h. *Initialize Array*: Cria um *array* do tipo da variável criada descrita no item **g.**, com a dimensão da constante numérica descrita em **a.**.

i. *Invoke Method Read*: Efetua a leitura de zeros elementos da FIFO "data".

j. *Greater Or Equal?*: Comparação que verifica se a quantidade de elementos remanescentes da FIFO "data", terminal *Elements Remaining*, são superiores ou igual aos do item **a.**.

k. *Invoke Method Read*: Efetua a leitura de 8000 elementos da FIFO "data".

l. *Replace Array Subset*: Substitui 8000 elementos do *array* inicializado fora do *loop*, conectado ao *shift register*.

m. Constante booleana: informa quando a RT FIFO "I&Q" pode receber os dados lidos da FIFO "data".

n. RT FIFO *Write*: Coloca os elementos na RT FIFO "I&Q".

o. Controle booleano "stop": Comanda o término do *loop* "read data".

p. *Invoke Method Stop*: Interrompe as FIFO's configuradas descritas na Figura 76.

q. *Close FPGA VI Reference*: Fecha a VI utilizada no *target*.

4.2.2.1 Objetivos

Leitura dos dados da FIFO "data", que contém os valores dos sinais I e Q demodulados. Transferência destes dados para o *loop* de mapeamento e correta finalização da execução das FIFO's.

4.2.2.2 Fluxo de dados

Dados de entrada:

- FIFO *Read* "data", variável do *array* de número tipo ponto fixo <+-20,4>;
- *Number Of Elements*, constante do tipo U32;
- Controle booleano "stop", variável do tipo booleana.

Dado de saída:

- RT FIFO "I&Q", variável do tipo *array* de número tipo ponto fixo <+-20,4>;

4.2.2.3 Otimizações realizadas

Criação do *array* de tamanho constante, 8000, através da estrutura *Initialize Array* para alocação fixa de memória. Desta maneira, o *array* é inserido como elemento inicial no *shift register* que realiza a transferência dos elementos da FIFO "data" para a RT FIFO "I&Q".

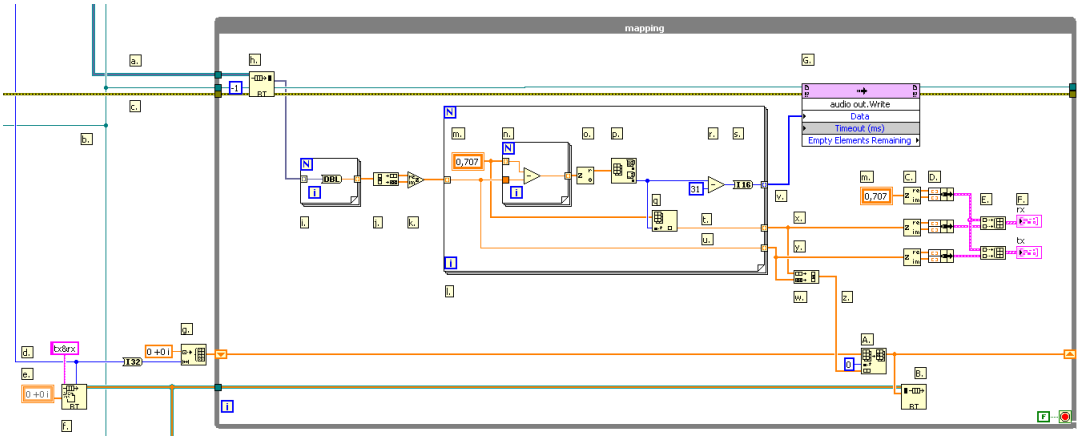
Leitura dos elementos da FIFO "data" através do método de elementos restantes. Primeiro lê-se zero elementos para verificar quantos restam. Até que o número de elementos na FIFO supere um valor de *threshold*, ela não é "esvaziada". Esta técnica permite ler um número constante de elementos mesmo com as flutuações de número de elementos entre o canal dos dois *buffers* do DMA.

4.2.2.4 Descrição do funcionamento

Este *loop* faz a leitura de zero elementos da FIFO e em seguida checa quantos elementos há na FIFO. Até que se atinja o valor limite pré-estabelecido, a FIFO permanece com todos seus elementos. Quando o valor de elementos for igual ou superior a 8000, é feita a leitura dos 8000 primeiros elementos, sendo que os elementos excedentes permanecem na FIFO até que as próximas iterações atinjam novamente o valor de 8000 ou mais. Os elementos retirados da FIFO são transferidos para o *array* que se encontra no *shift register* inicializado com a mesma quantidade de elementos para a leitura da FIFO. Estes são substituídos a cada leitura da FIFO. Somente quando a leitura da FIFO é feita os dados são passados para a RT FIFO "I&Q" para serem transferidos ao *loop* "mapping". Neste *loop* é feito o encerramento de todas as FIFO's ao se apertar o botão "stop". Caso não seja pressionado, na execução de uma nova abertura da VI, as FIFO's são resetadas através da configuração contida antes dos *loops* iniciarem.

4.2.3 Mapeamento de dados - *mapping loop*

Figura 79 – Diagrama de blocos do *loop* de mapeamento



Fonte: Produzido pelo autor

a. "Fio" de conexão para a RT FIFO "tx&rx".

b. "Fio" da referência da VI utilizada no FPGA.

c. "Fio" dos dados de erro: *Cluster* de três elementos, *status*, *code* e *source*.

d. "Fio" da constante numéria 8000.

e. *Array* do tipo *double complex*: Configura o tipo de dados que serão recebidos pela RT FIFO.

f. RT FIFO *Create*: Cria uma FIFO ""tx&rx" do tipo RT. Seus terminais são conectados aos itens descritos em **d.** e **e.**.

g. *Initialize Array*: Cria um *array* do tipo da variável criada descrita no item **e.**, com a dimensão da constante numérica descrita em **d.**.

h. RT FIFO *Read*: Recebe os elementos da RT FIFO "tx&rx" e efetua a sua leitura.

i. *Loop for*: Realiza a conversão dos dados, um a um, do tipo ponto fixo para *double*.

j. *Decimate 1D Array*: Separa os distintos *arrays* que foram inseridos de maneira a lê-los de maneira coerente.

k. *Re/Im To Complex*: Recebe as partes reais e imaginárias e as configura para um número do tipo complexo.

l. *For loop*: Realiza o mapeamento dos elementos recebidos.

m. *Array* constante: Possui os 64 valores do diagrama de constelação para a procura e comparação com os valores recebidos.

n. *For loop*: Subtrai, ponto a ponto, os elementos do diagrama de constelação com o valor recebido.

o. *Complex To Polar*: Recebe um número complexo e entrega o seu módulo e fase.

p. *Array Max & Min*: Encontra os valores máximos e mínimos de um *array*. Fornece sua localização e valor.

q. *Index Array*: Faz a seleção e leitura de um elemento de um *array*.

r. *Subtract*: Faz o decremento da constante numérica 31 do dado recebido da estrutura do item **p.**.

s. *To Word Integer*: Faz a conversão de um número do tipo I32 para outro do tipo I16.

t. "Fio" que seleciona o ponto do diagrama de constelação que foi recebido.

u. "Fio" com o valor do ponto recebido.

v. *Array* de dados que será enviado para a saída de áudio no FPGA.

x. *Array* de dados que foram mapeados na constelação.

y. *Array* de dados recebidos sem serem mapeados.

w. *Interleave 1D Arrays*: Agrupa dois *arrays*, onde o primeiro *array* ocupará as posições de ordem par e o segundo as posições de ordem ímpar.

z. *Array* de dados que serão transferidos para a RT FIFO "tx&rx" para o *loop* do cálculo do EVM e MER.

A. *Replace Array Subset*: Substitui 8000 elementos do *array* inicializado fora do *loop*, conectado ao *shift register*.

B. RT FIFO *Write*: Coloca os elementos na RT FIFO "tx&rx".

C. *Complex To Re/Im*: Recebe um número complexo e o configura para resultar em sua parte real e imaginária.

D. *Bundle*: agrupa dois *arrays*.

E. *Build Array*: Cria um *array* de dois *clusters*.

F. XY *Graphs*: Gráfico que recebe pontos com duas coordenadas (x,y).

G. *Invoke Method Write*: Envia um *array* de dados para o FPGA.

4.2.3.1 Objetivos

Com este *loop* tem-se o objetivo de fazer o mapeamento correto dos dados recebidos, isto é, fazer a escolha do ponto mais adequado para o conjunto de pares I e Q. Enviar o sinal recuperado para a saída de áudio analógica. Exibir o diagrama de constelação do sinal enviado e do sinal recebido. E, por fim, transferir os dados destes sinais para o *loop* que realiza o cálculo do EVM e MER.

4.2.3.2 Fluxo de dados

Dado de entrada:

- RT FIFO *Read* "I&Q", variável do tipo *array* de ponto fixo <+-20,4>;

Dados de saída:

- FIFO *Write* "audio out", variável do tipo *array* com dados do tipo I16;
- RT FIFO "tx&rx", variável do tipo *array* com dados do tipo *complex double*;
- Indicadores gráficos "tx" e "rx", variáveis gráficas do tipo *bundle* de dois *arrays* de números do tipo *double*.

4.2.3.3 Otimizações realizadas

Coloca-se o valor de *timeout* para a RT FIFO em -1, ou seja, a RT FIFO "I&Q" espera indefinidamente até que um dado esteja disponível para a leitura. Criação do *array* de tamanho constante, 8000, através da estrutura *Initialize Array* para alocação fixa de memória. Desta maneira, o *array* é inserido como elemento inicial no *shift register* que realiza a transferência do sinais transmitidos e recuperados para a RT FIFO "tx&rx".

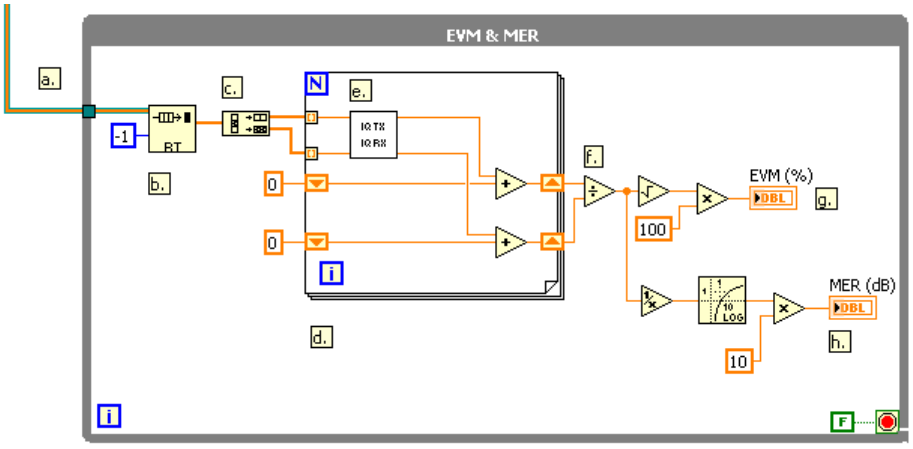
4.2.3.4 Descrição do funcionamento

Os dados recebidos pela RT FIFO "I&Q" são retirados da FIFO e convertidos para variáveis do tipo *double*, através de um *loop for* utilizando o conversor *To Double Precision Float*, por não ser possível converter todo *array* de uma só vez utilizando esse mesmo conversor. Após a conversão, os dados são colocados, de maneira sequencial, em um *Interleave Array*, o que faz com que haja a separação dos sinais I e Q na mesma ordem em que foram escritos no *loop for* presente no FPGA localizado no *loop* "to host". Como o sinal I constitui a parte real do número complexo e o sinal Q a sua parte imaginária, é utilizada a estrutura *Re/ImTo Complex* para criar este número complexo, para tirar proveito desta formatação. O *loop for* seguinte recebe os dados de maneira indexada, isto é, cada dado será lido e passado para outro *loop for* interno, onde ocorre a subtração

a todos elementos pertencentes ao mapa de constelação. Isto é feito para buscar dentre quais elementos há a menor diferença em módulo entre o sinal recebido e o sinal a ser assumido como recebido. Todos estas diferenças são armazenadas em um *array* onde será analisado pela estrutura *Array Max & Min* para que seja encontrada a menor diferença, ou seja, o valor mais próximo do sinal recebido. Tendo-se a localização deste elemento, equivalente ao índice, busca-se no mapa de constelação, *array* constante com os 64 valores distintos para esta modulação, e seleciona-se esse elemento para ser o dado escolhido como recebido. Diminui-se 31 deste número, pois é necessário para ter o sinal de volta ao seu nível original, que variava de -32 a +31, como descrito no *loop* de modulação no FPGA "modulation loop". Logo, este valor será o enviado ao FPGA para ser escrito na saída analógica de áudio. Como está operação é feita para todos elementos recebidos pelo *loop* de leitura de dados, cria-se um *array* de maneira indexada para que ao final do *loop for* externo, depois de convertidos para o valor de I16, aquele que recebe o sinal complexo, todos os dados tenham sido mapeados e possam ser enviados à FIFO "audio out". Também são armazenados neste *loop* os valores originais da recepção e os valores escolhidos no mapeamento após a recepção. Estes serão exibidos nos diagramas de constelação, "tx" e "rx". Também são transferidos para o *loop* "evm&mer" para o cálculo do erro do vetor de modulação e do erro de modulação, este último equivalente à relação sinal ruído para uma modulação digital. Esta transferência é feita através da criação de um *array* constante de 8000 elementos através da estrutura *Initialize Array* conectado a um *shift register*. A cada iteração subtitui-se estes elementos pelos elementos mapeados. São transferidos a RT FIFO "tx&rx" que será lida no *loop* mencionado anteriormente.

4.2.4 Cálculos do EVM e MER - EVM & MER *loop*

Figura 80 – Diagrama de blocos do *loop* que realiza o cálculo de EVM e MER



Fonte: Produzido pelo autor

a. "Fio" de conexão para a RT FIFO "tx&rx".

b. RT FIFO *Read*: Recebe os valores dos dados do *loop* de mapeamento.

c. *Decimate 1D Array*: Separa os distintos *arrays* que foram inseridos de maneira a lê-los de maneira coerente.

d. *Loop for*: Calcula os valores do numerador e denominador dos dados recebidos através da sub VI "evm".

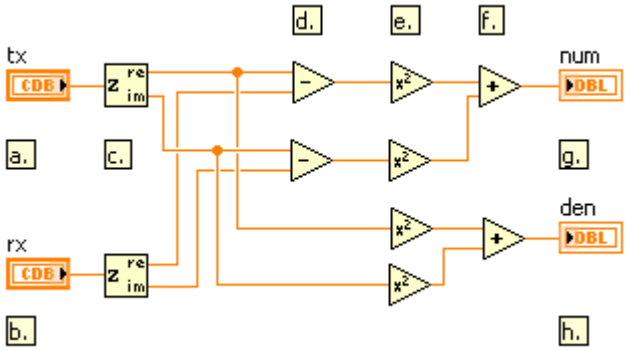
e. Sub VI "evm": Realiza o cálculo, ponto a ponto, do EVM.

f. *Divide*: Realiza a divisão entre os valores calculados para o numerador e denominador da sub VI "evm".

g. Indicador numérico "EVM (%)": Exibe o valor do EVM em %.

h. Indicador numérico "MER (dB)": Exibe o valor do MER em dB.

Figura 81 – Diagrama de blocos da sub VI "evm"



Fonte: Produzido pelo autor

a. Controle numérico "tx": Recebe o valor dos pontos do diagrama de constelação de acordo com o mapeamento.

b. Controle numérico "rx": Recebe o valor dos pontos do diagrama de constelação sem terem sido mapeados.

c. *Complex To Re/Im*: Recebe um número complexo e o configura para resultar em sua parte real e imaginária.

d. *Subtract*: Faz a subtração de dois números.

e. *Square*: Eleva um número ao quadrado.

f. *Add*: Soma dois números.

g. Indicador numérico "num": Exibe o valor do numerador.

h. Indicador numérico "den": Exibe o valor do denominador.

4.2.4.1 Objetivos

Realizar o cálculo do EVM e MER.

4.2.4.2 Fluxo de dados

Dado de entrada:

- FIFO "tx&rx", variável do tipo *complex double*.

Dados de saída:

- Indicadores numéricos "EVM (%)" e "MER (dB)", variáveis do tipo *double*.

4.2.4.3 Otimizações realizadas

Criou-se uma sub VI para agrupar os elementos que realizam os cálculos do erro do vetor de modulação e da relação sinal ruído para a modulação digital.

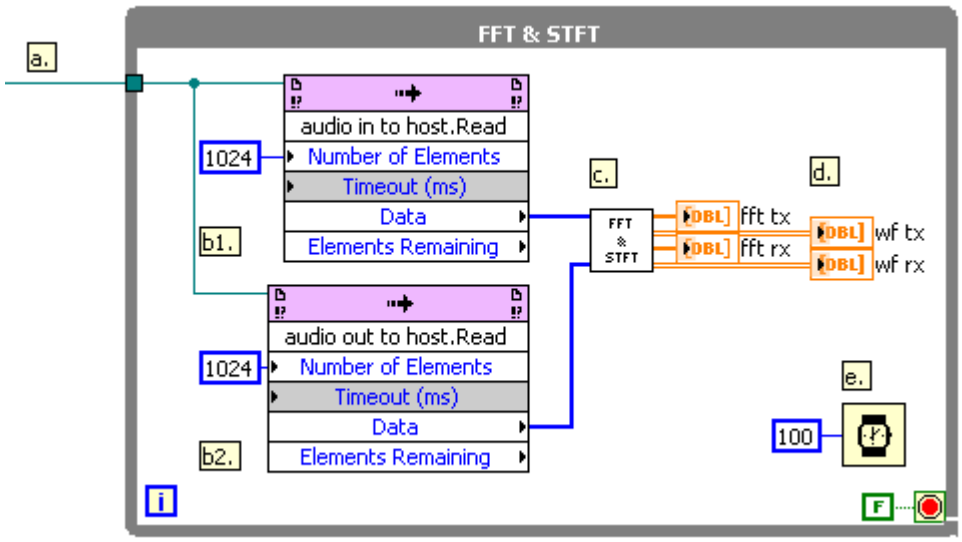
4.2.4.4 Descrição de funcionamento

Este *loop* recebe os dados dos pontos I e Q que foram recebidos originalmente e os pontos escolhidos para a sua representação no diagrama de constelação. Os recebidos originalmente compõem a base para comparação com os que foram mapeados, calculando-se assim a sua diferença.

O *array* de dados é recebido pelo RT FIFO *Read* "tx&rx" e então separado em *arrays* dos pontos dos sinais I e Q. Através de uma indexação de ambos os vetores, cada ponto será escrito nos conectores de entrada da sub VI "evm" e como saída desta, será obtido os valores para o numerador e denominador do EVM. Esses resultados são somados aos resultados de iterações anteriores através de um *shift register*, para somatório dos dados. Isto será feito para todo o vetor de dados recebidos do *loop* "mapping". Ao final deste *loop*, realizam-se os cálculos finais para ser obtido os valores dos dois parâmetros de interesse.

4.2.5 Cálculo da FFT e STFT - FFT & STFT *loop*

Figura 82 – Diagrama de blocos do *loop* "FFT & STFT"



Fonte: Produzido pelo autor

a. "Fio" da referência da VI utilizada no FPGA.

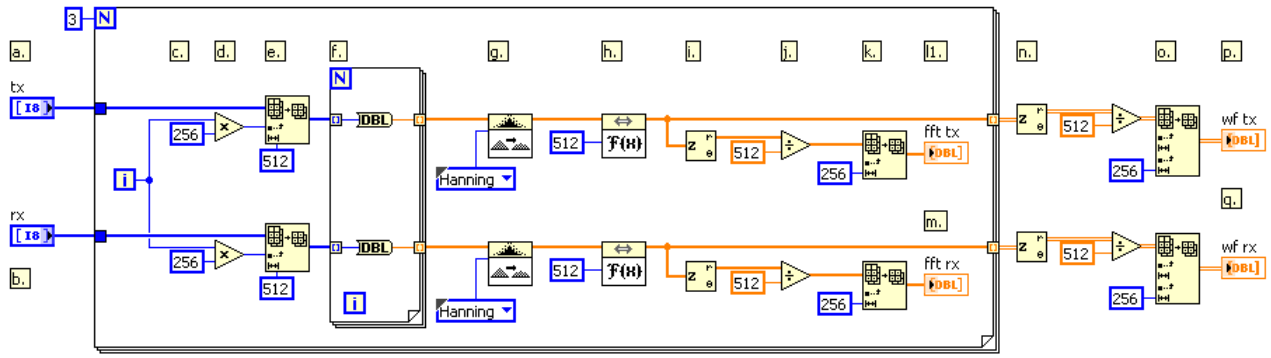
b1. e b2. *Invoke Method Read*: Fazem a leitura de 1024 elementos das FIFO's "audio in to host" e "audio out to host".

c. Sub VI "fft&stft": Sub VI que realiza o cálculo da FFT e da STFT dos sinais de entrada e saída de áudio.

d. Indicadores gráficos "fft rx", "fft tx", "wf rx" e "wf tx": Gráficos que exigem os resultados da FFT e STFT para os sinais de entrada e saída de áudio.

e. *Wait* (ms): Sub VI que introduz um tempo de espera de 100 ms entre iterações.

Figura 83 – Diagrama de blocos da sub VI “fft”.



Fonte: Produzido pelo autor

a. Indicador *array* numérico “tx”: Recebe os valores lidos pela FIFO “audio out to host”.

b. Indicador *array* numérico “rx”: Recebe os valores lidos pela FIFO “audio in to host”.

c. Constante numérica 256.

d. *Multiply*: Faz a multiplicação entre o valor da iteração do *loop* e a constante 256.

e. *Array Subset*: Faz a leitura de uma porção de um *array*. Configura-se a posição inicial de leitura e sua extensão.

f. *For loop*: Realiza a conversão de um número inteiro de 8 *bits* em um do tipo *double*.

g. Sub VI “Scaled Time Domain Window”: Aplica um janelamento do tipo Hanning para o sinal de entrada conectado ao seu terminal.

h. Sub VI “FFT”: Calcula a FFT do sinal de entrada conectado ao seu terminal.

i. *Complex To Polar*: Exibe o valor dos módulos e fases do *array* de números complexos de entrada.

j. Divide: Realiza a divisão entre os valores resultantes da FFT e da constante numérica 512.

k. *Array Subset*: Item **e.**.

l1. e m. Gráficos das FFT’s dos sinais recebidos e transmitidos.

n. *Complex To Polar*: Recebe um *array* de números complexos e entrega os seus módulos e fases.

o. *Array Subset*: Item **e.**.

p. e q. Gráficos das STFT dos sinais recebidos e transmitidos.

4.2.5.1 Objetivos

Realizar o cálculo da FFT e da STFT dos sinais enviados e recebidos pela saída de áudio.

4.2.5.2 Fluxo de dados

Dados de entrada:

- FIFO's "audio in to host" e "audio out to host", variáveis do tipo I8.

Dados de saída:

- Indicadores gráficos "fft tx", "fft rx", "wf tx" e "wf rx", variáveis do tipo *double*.

4.2.5.3 Descrição de funcionamento

Os dados dos sinais de entrada e saída de áudio são lidos das FIFO's que os escrevem no FPGA. Como aqui não se tem preocupação com a eventual perda de dados contidos nas FIFO's "audio in to host" e "audio out to host", não é utilizada a mesma estratégia que foi exposta para o *loop* "read data". Assim, a cada execução deste *loop* são lidos 1024 elementos de cada FIFO. Os dados lidos são encaminhados para um *loop for* sem auto-indexação. Este *loop* executará três vezes por conta do algoritmo de janelamento escolhido. Para a primeira iteração serão lidos os índices 0 a 511, na segunda 256 a 767 e na terceira 768 a 1023. Isto é feito utilizando-se a estrutura *Array Subset*, utilizando-se em seu terminal o produto do controle numérico que exige a iteração atual do *loop* e a constante numérica 256. Com essa atividade será gerado um *array* de 512 elementos que será convertido no *loop* for através da estrutura *To Double Precision Float*. Após a conversão é feito o janelamento dos dados através da sub VI *Scaled Time Domain Window* com a escolha da janela do tipo Hanning. Em seguida utiliza-se a sub VI FFT que realiza a transformada rápida de Fourier no *array*. Configura-se em seu terminal de entrada a quantidade de 512 elementos para o correto processamento. Estes dados são conectados ao extremo final do *loop for* para que sejam concatenados, isto é, cada conjunto de dados dos *arrays* sejam escritos um após o outro na ordem em que são criados, pois serão utilizados para o gráfico *Intensity Chart* responsável pela visualização do diagrama de cascata. Para cada iteração do *loop* são mostrados os valores para o resultado da FFT através dos gráficos "fft tx" e "fft rx". Para correta visualização dos dados, utiliza-se a estrutura *Complex To Polar* para o cálculo dos módulos dos números complexos contidos no *array* de entrada, estes são divididos por 512 e conectados à estrutura *Array Subset*, selecionando-se 256

elementos para a leitura. Após o término do *loop for*, os *arrays* concatenados são passados pelo mesmo processo feito para a visualização dos dados da FFT.

5 Análise de desempenho

Como resultado da implementação adotada, foi possível comprovar a correta operação da arquitetura de SDR desenvolvida. Para uma melhor visualização da qualidade da transmissão e recepção, serão apresentados os gráficos da FFT, STFT e do diagrama de constelação para a adotada (64QAM) a partir de um sinal multi-tonal com componentes de 1 kHz, 5 kHz, 7.5 kHz e 10 kHz. A fim de visualizar a adição do ruído ao sinal, a amplitude do ruído de tipo branco gaussiano foi incremental de maneira exposta na Tabela 2, onde também são apresentados os valores para o MER e EVM.

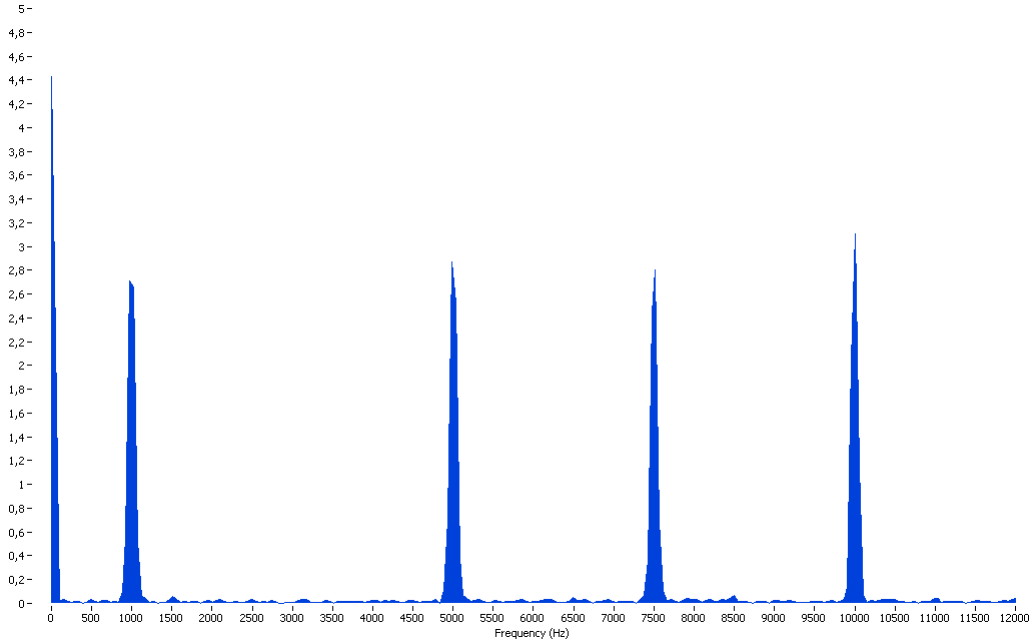
Tabela 2 – Valores do MER e EVM em função do nível de ruído

σ	MER (dB)	EVM (%)
0	40	0,9
50	25	5
100	20,5	9,2
150	18,3	12
300	16	15,6

O σ do ruído corresponde a entrada da Sub VI "Gaussian White Noise"e é descrito conforme (NATIONAL INSTRUMENTS, 2012).

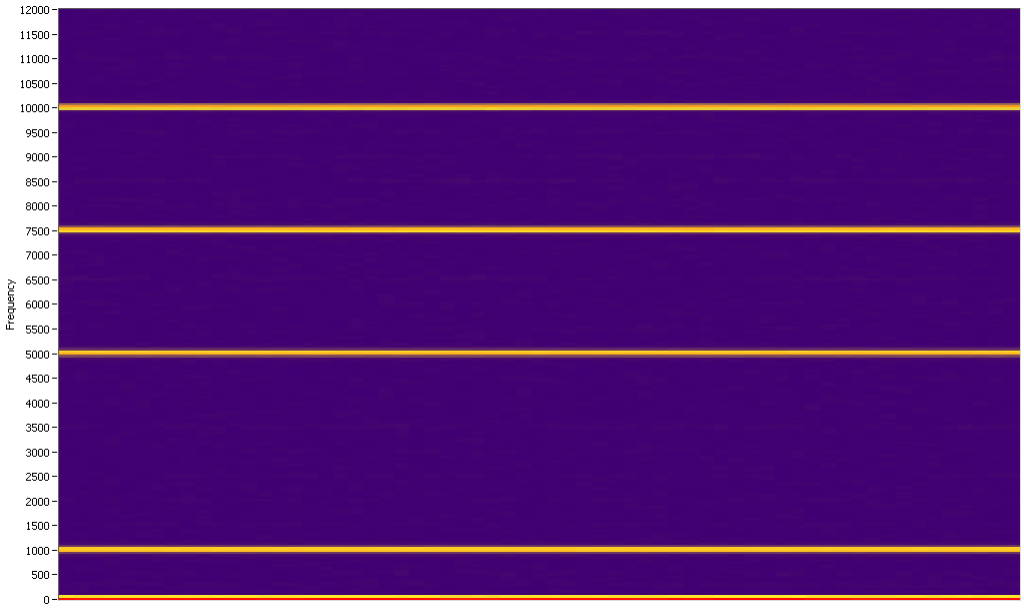
Para o sinal transmitido tem-se as Figuras 84, 85 e 86 para a FFT, a STFT e o diagrama de constelação, respectivamente. Verifica-e que, sem adição de ruído, o sinal recebido, apresentado nas Figuras 87, 88 e 89, é recuperado com um EVM de 0,9% (Tabela 2). Aumentando-se σ para 50, o EVM aumenta para 5% e seu MER diminui de 40 para 25 (dB), conforme o diagrama de constelação da Figura 90. Nesta configuração o ruído ainda não é possível de ser visualizado no espectro de frequências. Para $\sigma = 100$, tem-se o resultado nas Figuras 91, 92 e 93, com MER a 20,5 dB e EVM em 9,2%. Para este valor de σ já é possível visualizar a presença do ruído nos gráficos da FFT e STFT, assim como se torna mais acentuado no diagrama de constelação, comparado com o caso anterior. Com σ em 150 o diagrama de constelação fica ainda mais deteriorado, onde o EVM atinge 12% e com σ em 300 passa para 15,6%, assim como o BER diminui de 18,3 para 16 dB.

Figura 84 – FFT para o sinal transmitido



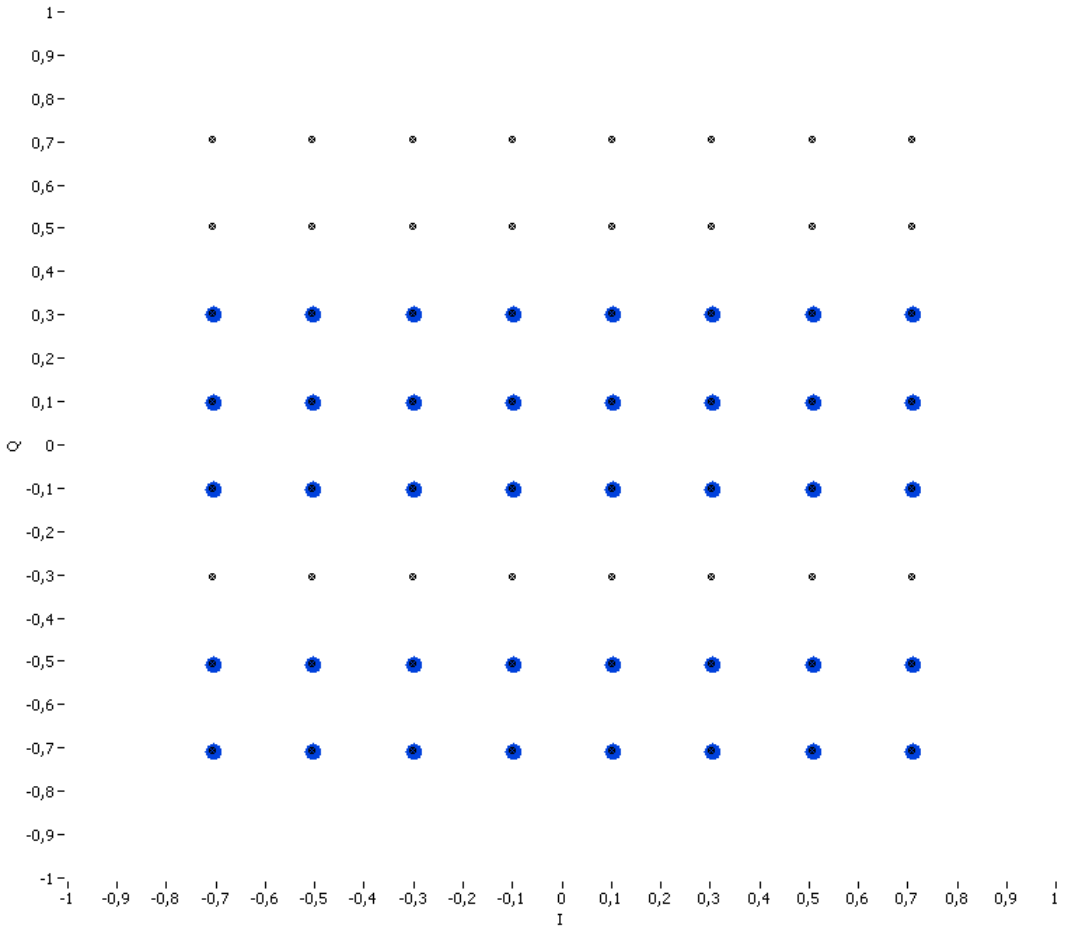
Fonte: Produzido pelo autor

Figura 85 – STFT para o sinal transmitido



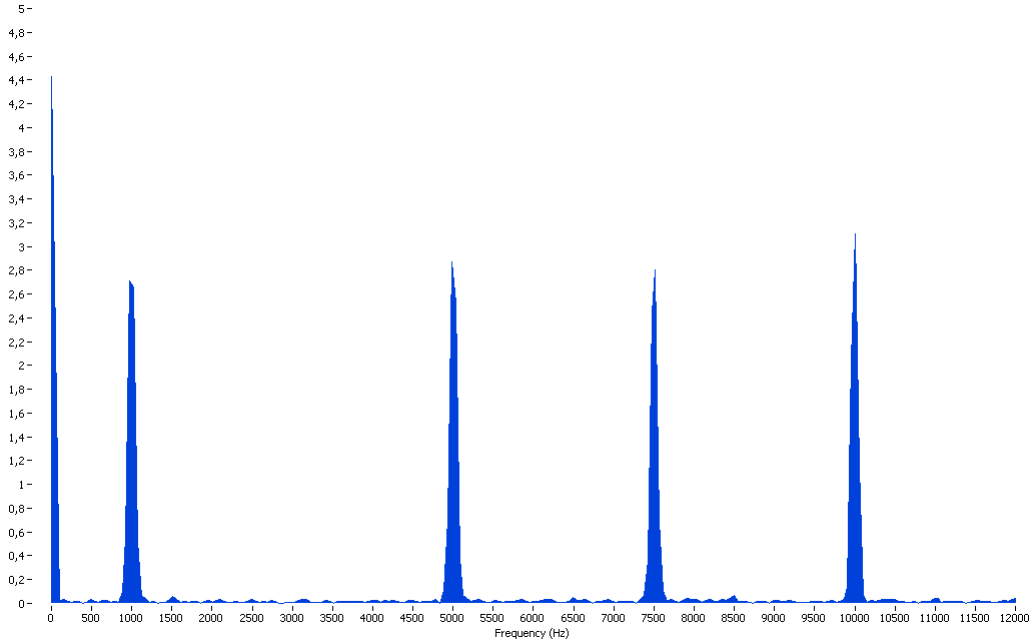
Fonte: Produzido pelo autor

Figura 86 – Diagrama de constelação para o sinal transmitido



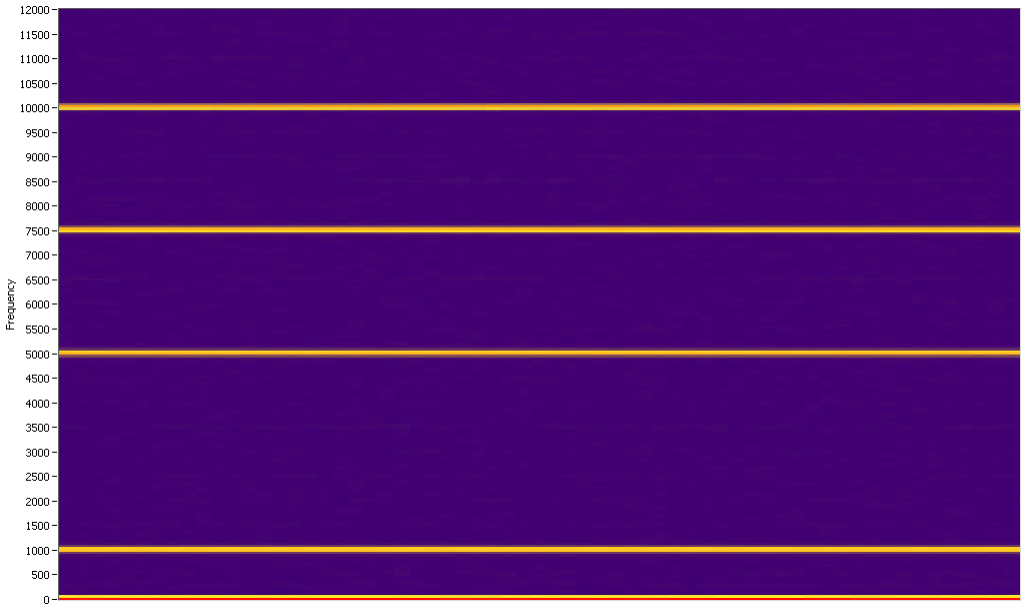
Fonte: Produzido pelo autor

Figura 87 – FFT para o sinal recebido sem adição de ruído



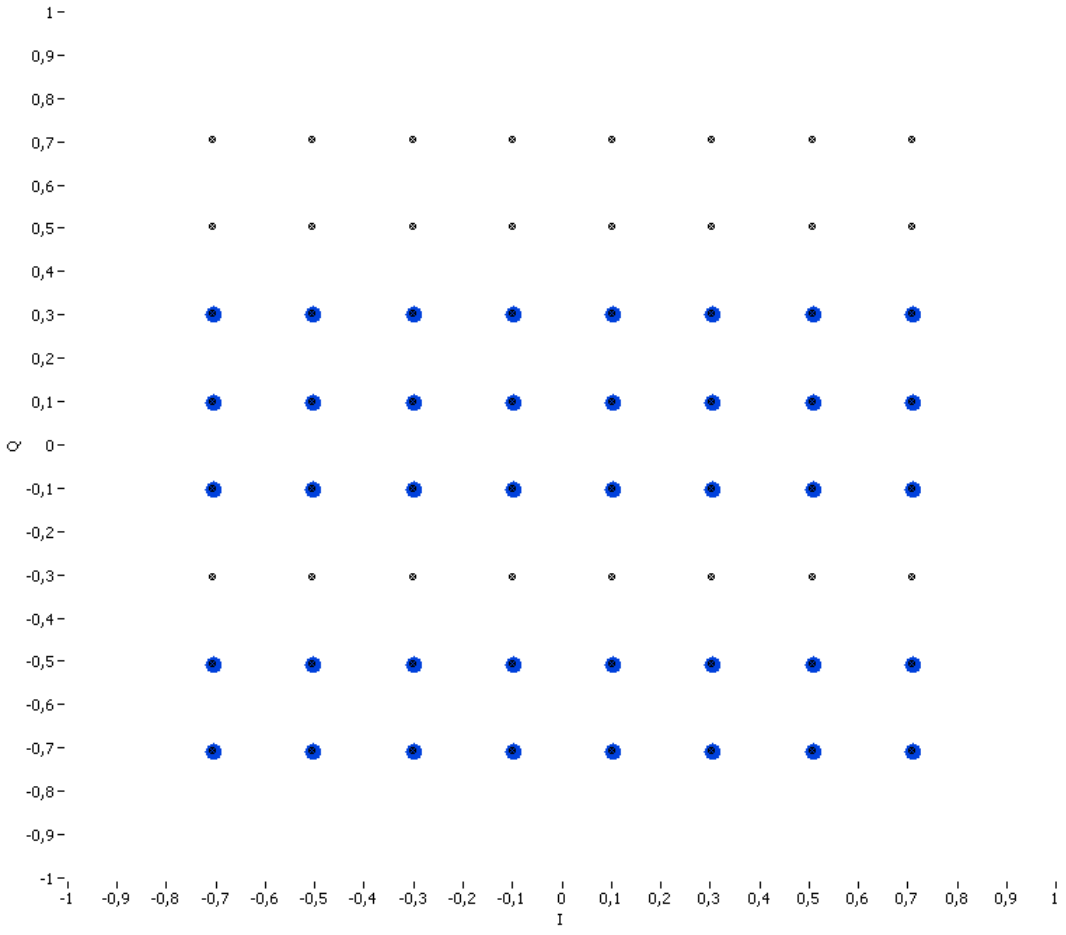
Fonte: Produzido pelo autor

Figura 88 – STFT para o sinal recebido sem adição de ruído



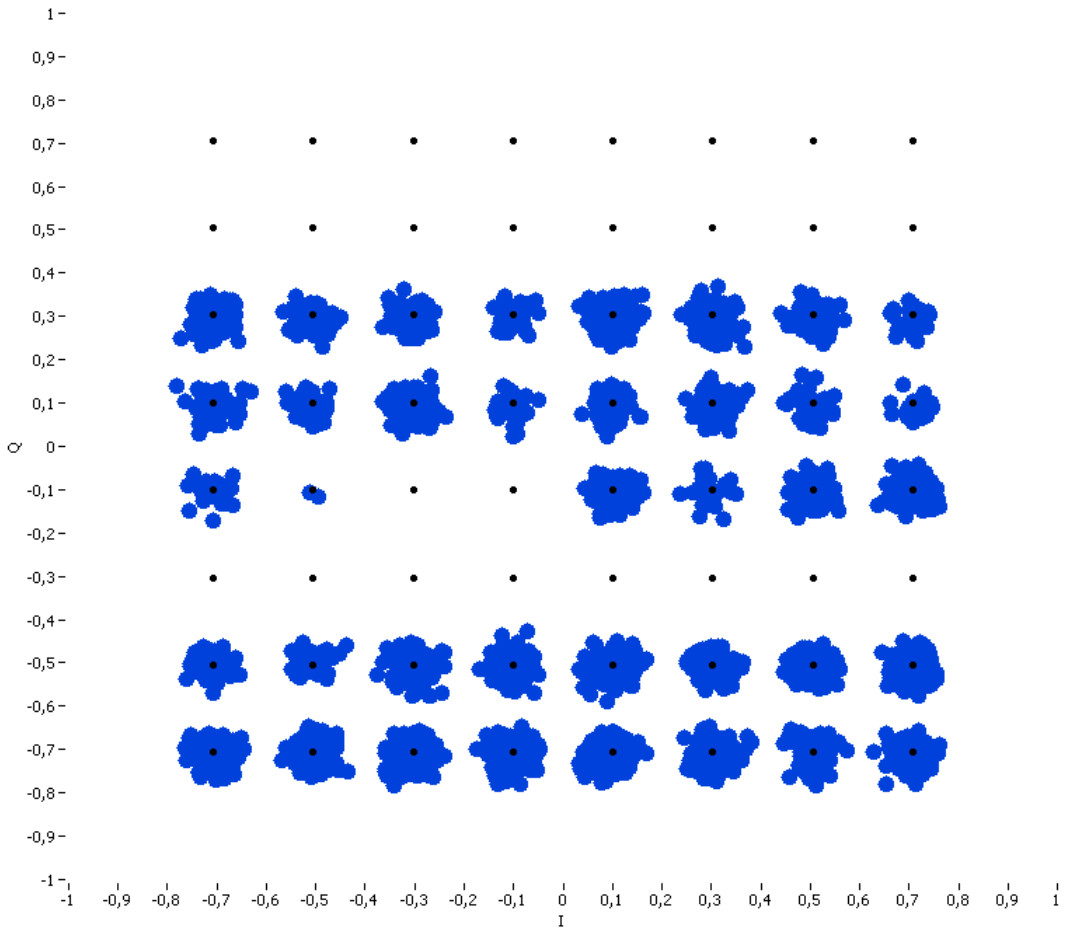
Fonte: Produzido pelo autor

Figura 89 – Diagrama de constelação para o sinal recebido sem adição de ruído



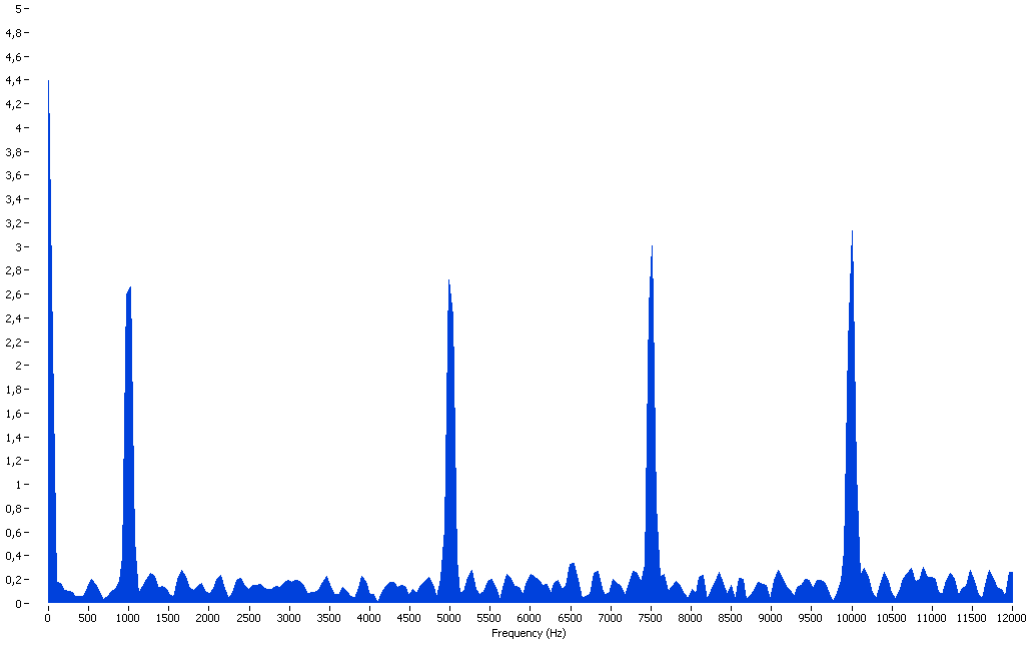
Fonte: Produzido pelo autor

Figura 90 – Diagrama de constelação para o sinal recebido com a adição de ruído($\sigma = 50$)



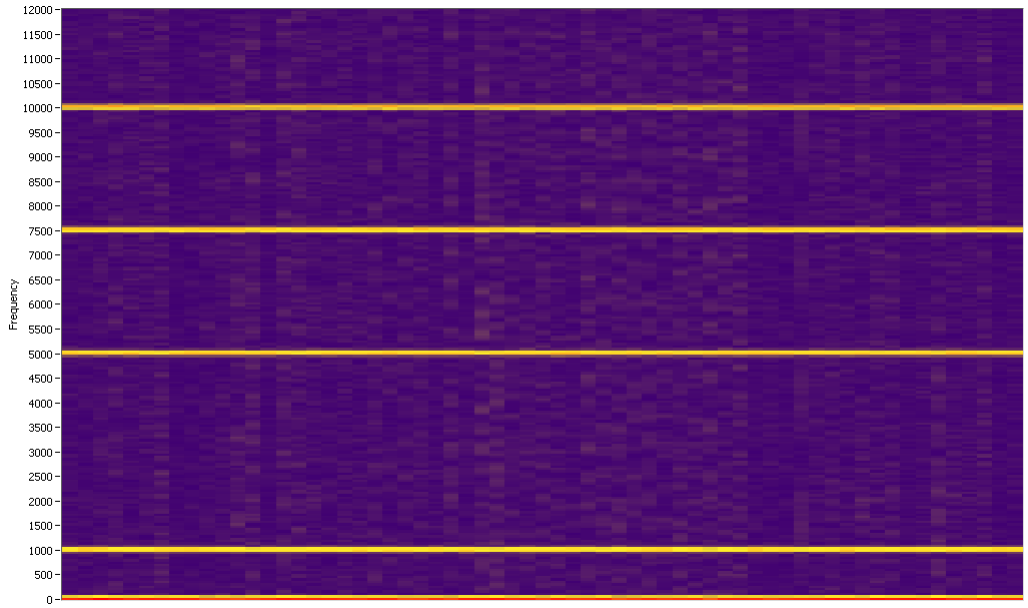
Fonte: Produzido pelo autor

Figura 91 – FFT para o sinal recebido com adição de ruído ($\sigma = 100$)



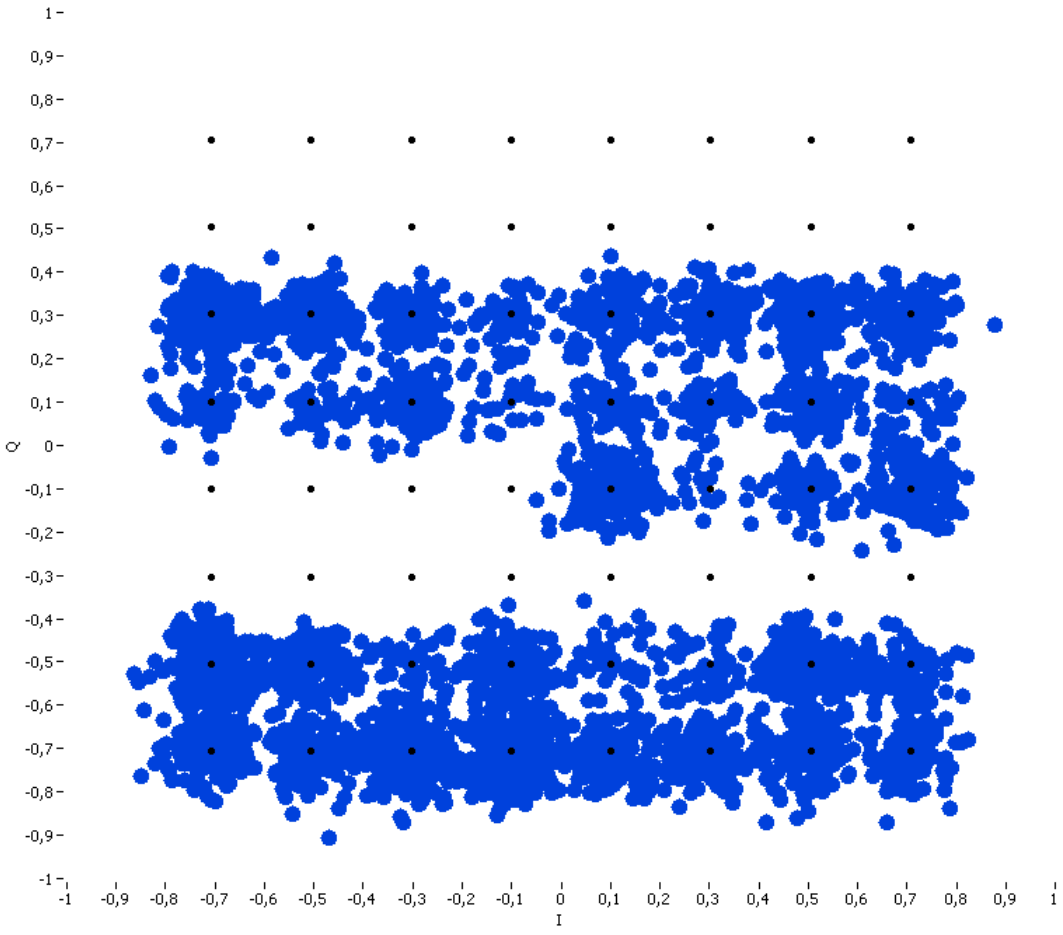
Fonte: Produzido pelo autor

Figura 92 – STFT para o sinal recebido com a adição de ruído($\sigma = 100$)



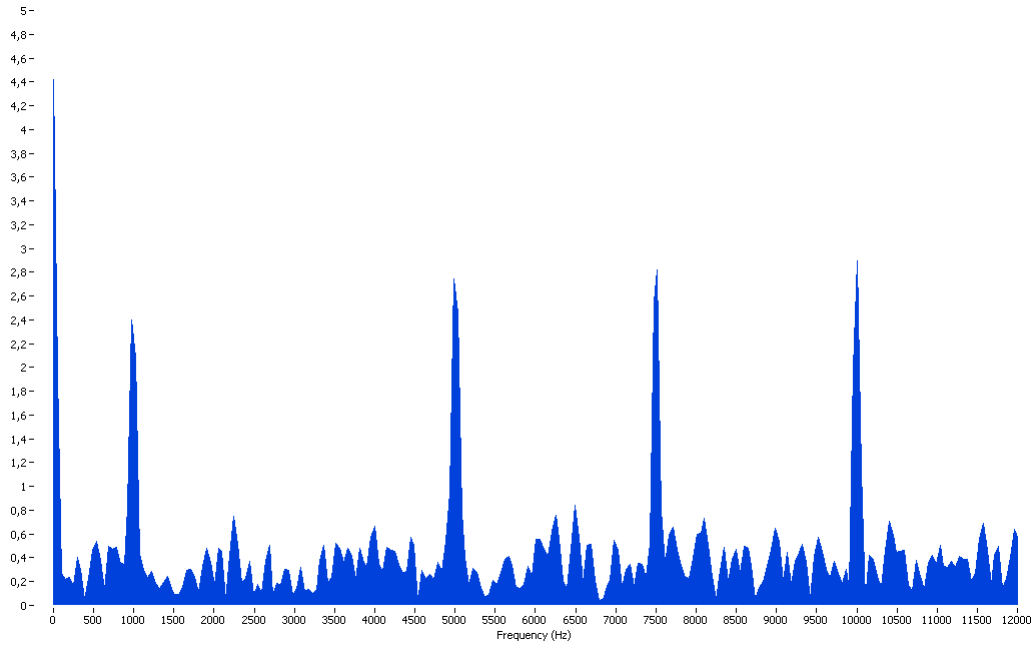
Fonte: Produzido pelo autor

Figura 93 – Diagrama de constelação para o sinal recebido com a adição de ruído($\sigma = 100$)



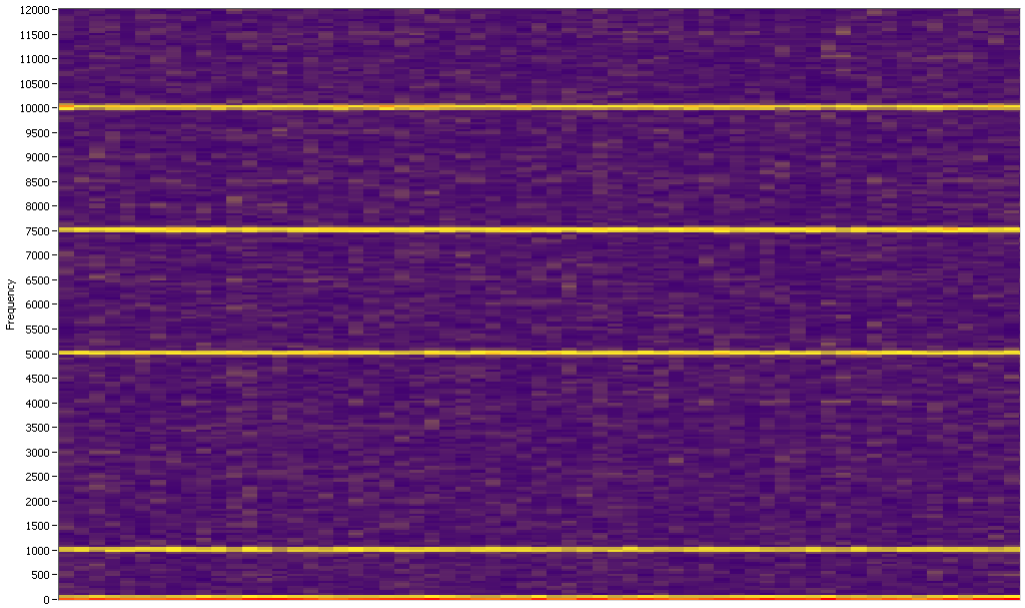
Fonte: Produzido pelo autor

Figura 94 – FFT para o sinal recebido com adição de ruído ($\sigma = 150$)



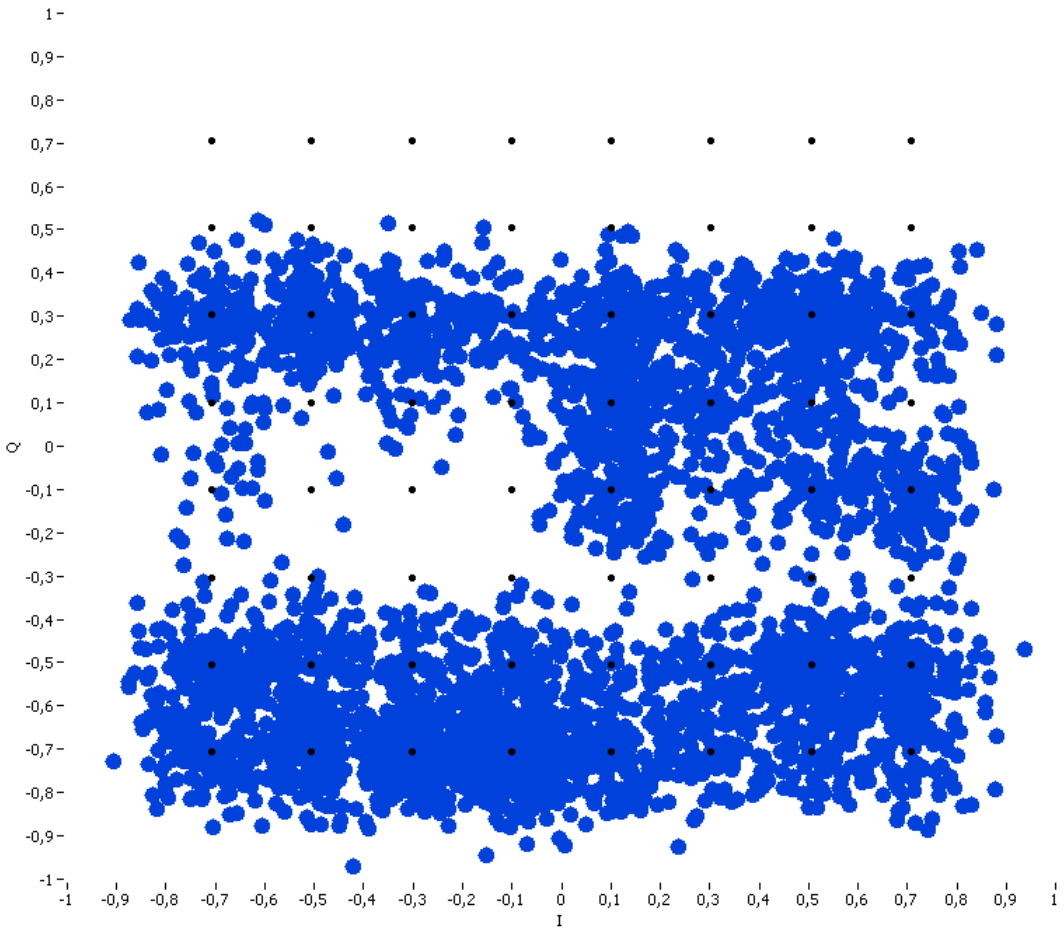
Fonte: Produzido pelo autor

Figura 95 – STFT para o sinal recebido com a adição de ruído($\sigma = 150$)



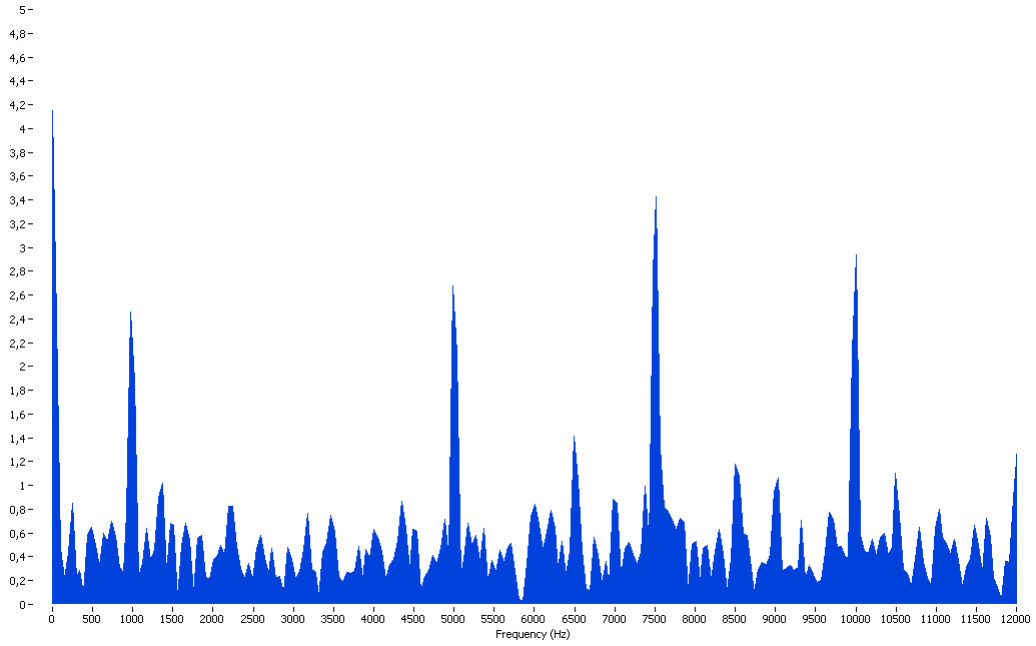
Fonte: Produzido pelo autor

Figura 96 – Diagrama de constelação para o sinal recebido com a adição de ruído($\sigma = 150$)



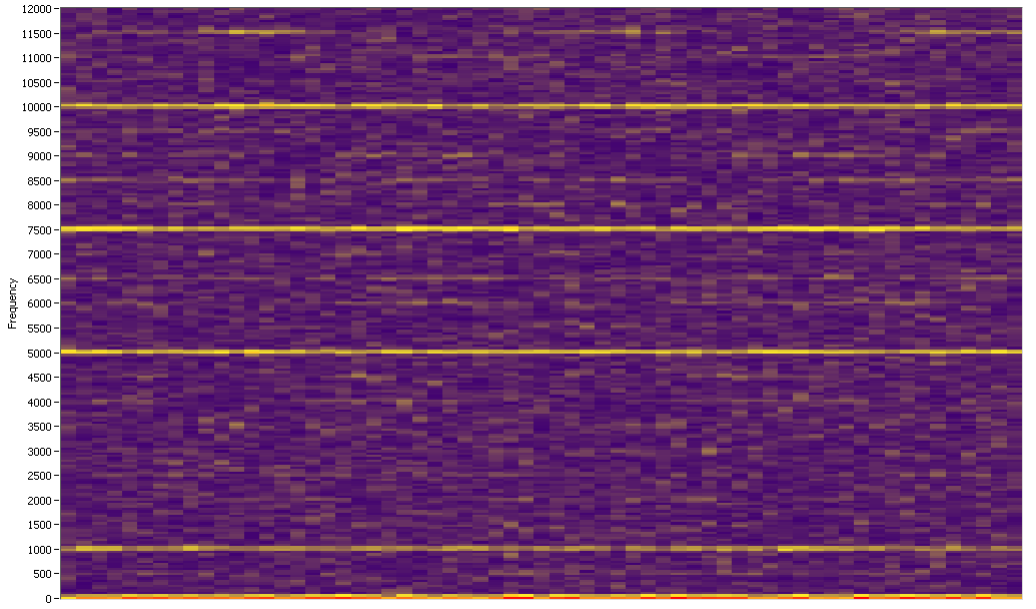
Fonte: Produzido pelo autor

Figura 97 – FFT para o sinal recebido com adição de ruído ($\sigma = 300$)



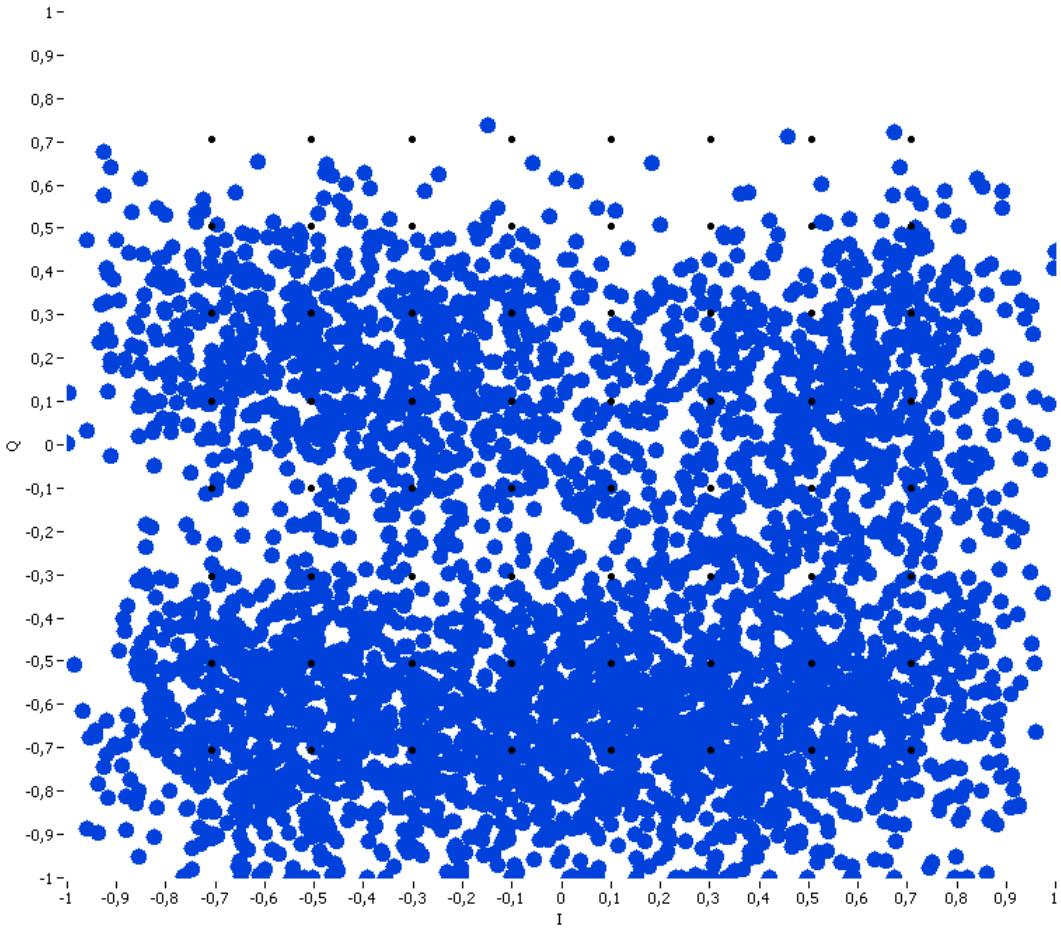
Fonte: Produzido pelo autor

Figura 98 – STFT para o sinal recebido com a adição de ruído($\sigma = 300$)



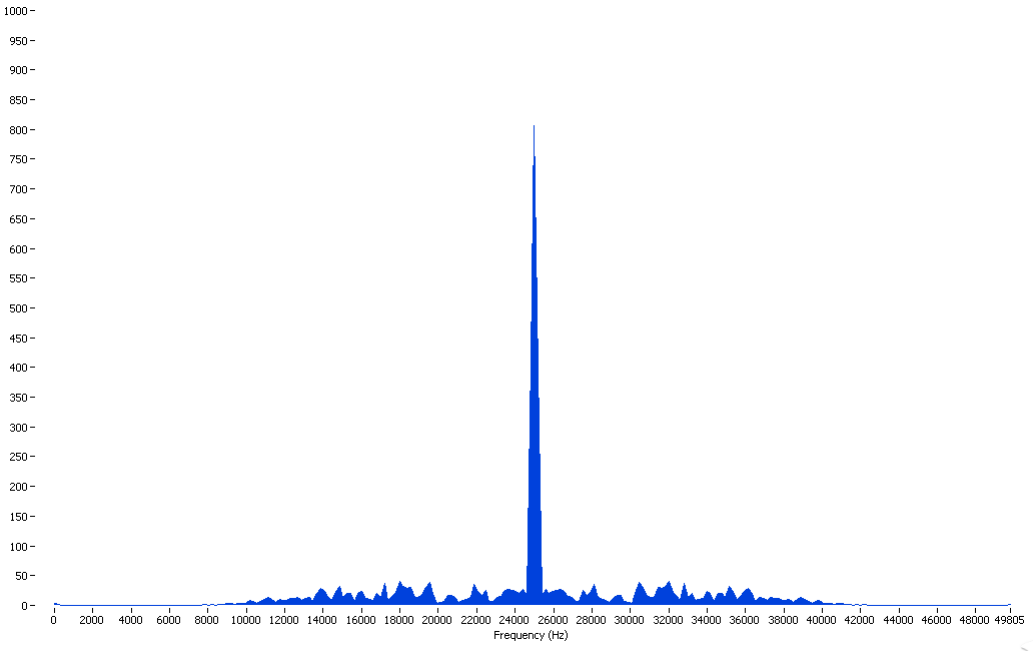
Fonte: Produzido pelo autor

Figura 99 – Diagrama de constelação para o sinal recebido com a adição de ruído($\sigma = 300$)



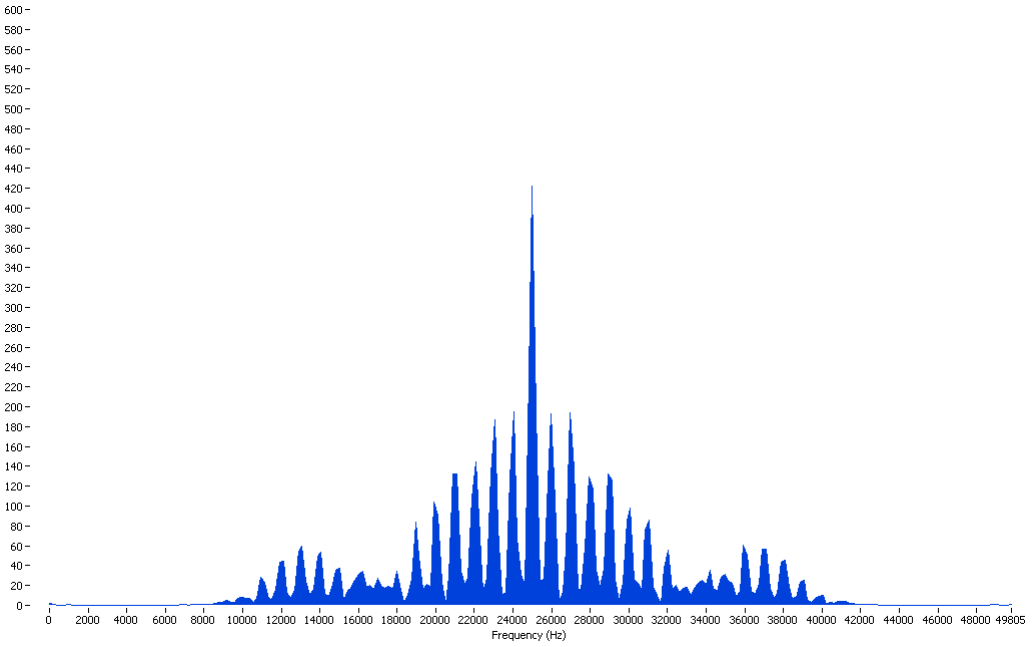
Fonte: Produzido pelo autor

Figura 100 – FFT do sinal transmitido pelo conector analógico



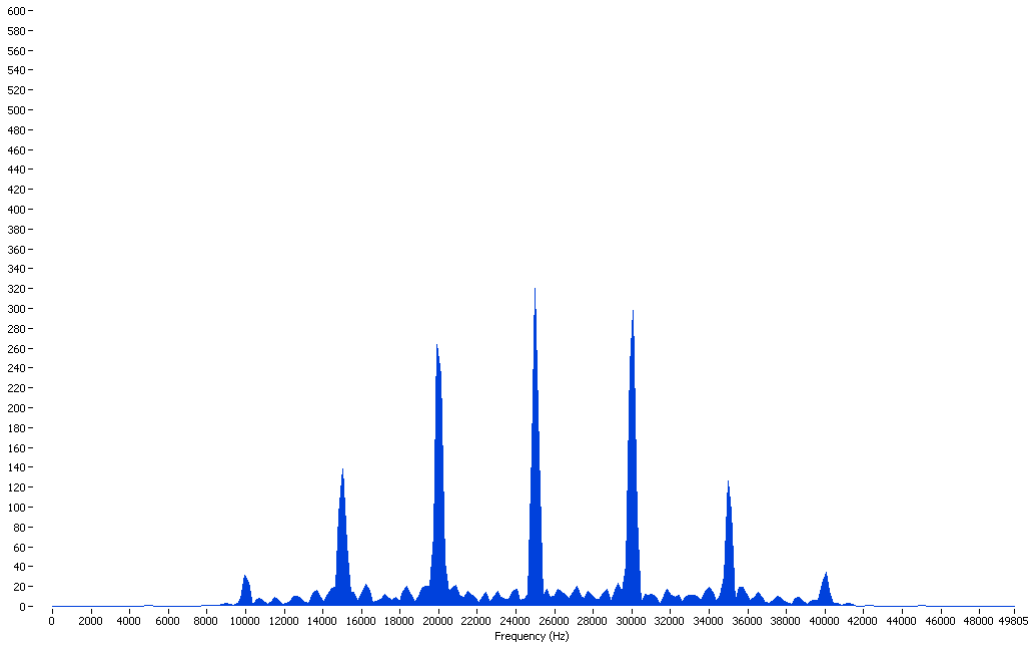
Fonte: Produzido pelo autor

Figura 101 – FFT do sinal transmitido pelo conector analógico com um sinal tonal de 1 kHz



Fonte: Produzido pelo autor

Figura 102 – FFT do sinal transmitido pelo conector analógico com um sinal tonal de 5 kHz



Fonte: Produzido pelo autor

Na Figura 100 fica evidente a presença do sinal da portadora de 25 kHz, visto que o sinal que compõe a portadora é de 4 pontos e que a frequência do *loop* de transmissão opera é de 100 kHz. Na Figura 101 e 102 é possível ver como o sinal modulante altera a frequência do sinal transmitido.

Como último resultado é mostrado na Tabela 3 os dados da compilação da VI presente no *target*.

Tabela 3 – Dados da compilação

Device Utilization	Used	Total	Percent
Slice Registers	17126	35200	48,7
Slice LUT's	15316	17600	87,0
Block RAM's	27	60	45
DSP48's	78	80	97,5

6 Conclusões e trabalhos futuros

Verificou-se que, através da escolha da plataforma heterogênea utilizada LabVIEW FPGA com RT, foi possível construir um rádio que possui todas as suas funções operativas implementadas em *software*. Foi possível implementar na totalidade todas as funções mais importantes, isto é, aquelas que requerem mais capacidade de processamento, amostragem do sinal de áudio, *upsampling*, modelagem de pulso, modulação, transmissão e recepção, demodulação e *downsampling* no FPGA. Assim, obteve-se um sistema totalmente síncrono, com a precisão necessária para toda as operações digitais. Para o monitoramento dos sinais de interesse houve duas situações, as que não poderiam ocorrer perda de informações e as que poderiam. A primeira sendo a transferência do sinal demodulado para o RT, onde busca-se a recuperação da informação recebida. Na segunda, remete-se ao caso da amostragem dos sinais de entrada e saída de áudio, onde não havia necessidade de ter uma análise em tempo real para o cálculo da FFT e STFT. Em ambos os casos foi possível atingir o objetivo, através da correta configuração das FIFO's do canal DMA, mostrando que estas funcionalidades favorecem uma alta capacidade de troca de dados entre plataformas de comando distintas. Os recursos presentes no dispositivo utilizado, myRIO, tornaram a experiência deste estudo ainda mais didática por dispor das entradas e saídas de áudio, pois assim foi possível escutar e perceber as características do áudio recebido em comparação ao transmitido. A partir dos testes com a inserção de ruído foram calculados a relação sinal ruído para este tipo de transmissão (digital) e também o erro presente na localização do ponto recebido no diagrama de constelação. Sua visualização torna-se muito importante, pois é de onde extraem-se características físicas do sistema através de gráficos, mostrando ao usuário de uma maneira simples e direta como o sistema está operando. Foi uma oportunidade de entender como os sistemas de comunicação funcionam e também encontrar pontos não tão satisfatórios do sistema, onde melhorias podem ser feitas em trabalhos futuros.

A continuidade do trabalho pode ser dada de muitas maneiras, como por exemplo, a adição de mais tipos de modulação de amplitude em quadratura, para ser possível obter uma comparação da robustez quanto ao nível da relação sinal ruído para outros casos, 4QAM, 16QAM, 32 QAM, e assim por diante; um novo projeto para a relação entre o *downsampling* e *upsampling* juntamento com a modelagem de pulso e da transformada de Hilbert, ambos implementados por filtros do tipo FIR. No projeto atual, utilizou-se os propostos em (KEHTARNAVAZ; MAHOTRA, 2010), e assim, manteve-se o sistema com características semelhantes, no entanto, de acordo com os resultados, foram satisfatórios para esta proposta. E por fim, o projeto do *front-end* do rádio para a tentativa de faze-lo operar nas frequências de RF. Sugere-se não utilizar a saída e entrada analógica do

conector C, devido a limitação de seus conversores operarem até uma taxa de 345kHz (NATIONAL INSTRUMENTS, 2013), e sim as entradas e saídas digitais dos conectores A e B juntamente com um ADC e um DAC conectados a este. Os modelos pesquisados correspondem as componentes ADS807 e DAC902U, ambos da *Texas Instruments*.

.

Referências

CROCHIERE, R. E.; RABINER, L. R. *Multirate Digital Signal Processing*. Englewood Cliffs, NJ, USA: Pretince Hall, 1983. Citado na página 60.

HAHN, S. L. *Hilbert Transforms in Signal Processing*. Norwood, MA, USA: Artech House, 1996. Citado na página 67.

HAYKIN, S. *Digital Communications Systems*. Hoboken, NJ, USA: Wiley, 2014. Citado na página 65.

HAYKIN, S.; VEEN, B. V. *Signals and Sistems*. Hoboken, NJ, USA: Wiley, 2003. Citado na página 64.

HEIM, A. *Make it Faster: More Throughput or Less Latency?* 2014. Disponível em: <http://www.ni.com/white-paper/14990/en/>. Citado 2 vezes nas páginas 48 e 49.

IP Corner: The LabVIEW Fixed-Point Data Type Part 2 – Working with Fixed-Point. 2009. Disponível em: <http://www.ni.com/newsletter/50363/en/>. Citado na página 90.

KEHTARNAVAZ, N.; MAHOTRA, S. *Digital Signal Processing Laboratory: LabVIEW-Based FPGA Implementation*. Boca Raton, FL, USA: Brown Walker Press, 2010. Citado 4 vezes nas páginas 21, 51, 52 e 133.

LATHI, B. P. *Modern Digital and Analog Communication Systems (The Oxford Series in Electrical and Computer Engineering)*. New York, NY, USA: Oxford University Press, 1998. Citado na página 65.

MITOLA, J. Software radios - survey, critical evaluation and future directions. *National Telesystems Conference*, v. 3, n. 1, p. 13/15–13/23, 1992. Citado na página 21.

NATIONAL INSTRUMENTS. *CompactRIO Developers Guide*. [S.l.], 2009. Disponível em: <http://www.ni.com/pdf/products/us/fullcriodevguide.pdf>. Citado na página 27.

NATIONAL INSTRUMENTS. *LabVIEW 2012 Help*. 2012. Disponível em: <http://zone.ni.com/reference/en-XX/help/371361J-01/>. Citado 6 vezes nas páginas 27, 34, 35, 36, 42 e 119.

NATIONAL INSTRUMENTS. *User Guide and Specifications NI myRIO 1900*. [S.l.], 2013. Disponível em: <http://www.ni.com/pdf/manuals/376047a.pdf>. Citado 2 vezes nas páginas 24 e 134.

NATIONAL INSTRUMENTS. *NI LabVIEW High - Performance FPGA Developer's Guide*. [S.l.], 2014. Disponível em: <http://download.ni.com/pub/gdc/tut/labview_high-perf_fpga_v1.1.pdf>. Citado 3 vezes nas páginas 27, 35 e 37.

OPPENHEIM, A. V.; SCHAER, R. W. *Discrete-time Signal Processing*. New Jersey, NJ, USA: Pretince Hall, 2010. Citado 6 vezes nas páginas 57, 58, 59, 69, 70 e 71.

PROAKIS, J. G. *Digital Communications*. New York, NY, USA: McGraw-Hill, 2008. Citado 6 vezes nas páginas 53, 54, 56, 65, 66 e 67.

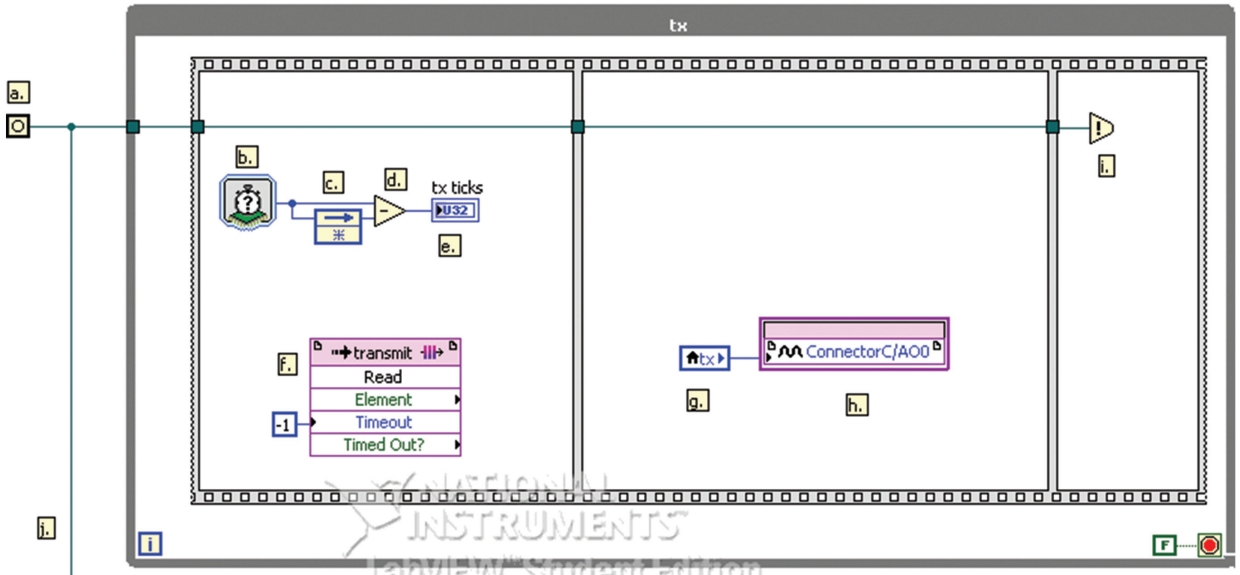
REED, J. H. *Software Radio: A Modern Approach to Radio Engineering*. Uper Saddle River, NJ, USA: Prentice Hall, 2002. Citado na página 51.

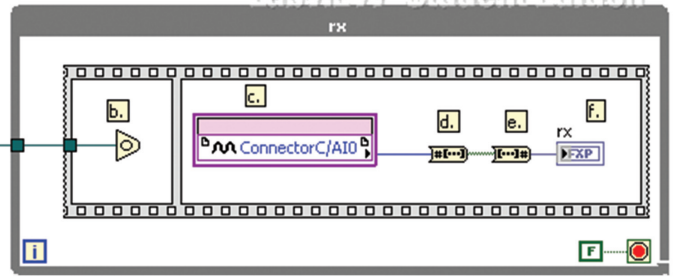
SDR FORUM. *SDRF Cognitive Radio Definitions*. 2007. Disponível em: <http://www.sdrforum.org/pages/documentLibrary/documents/SDRF-06-R-0011-V1_0_0.pdf>. Citado na página 21.

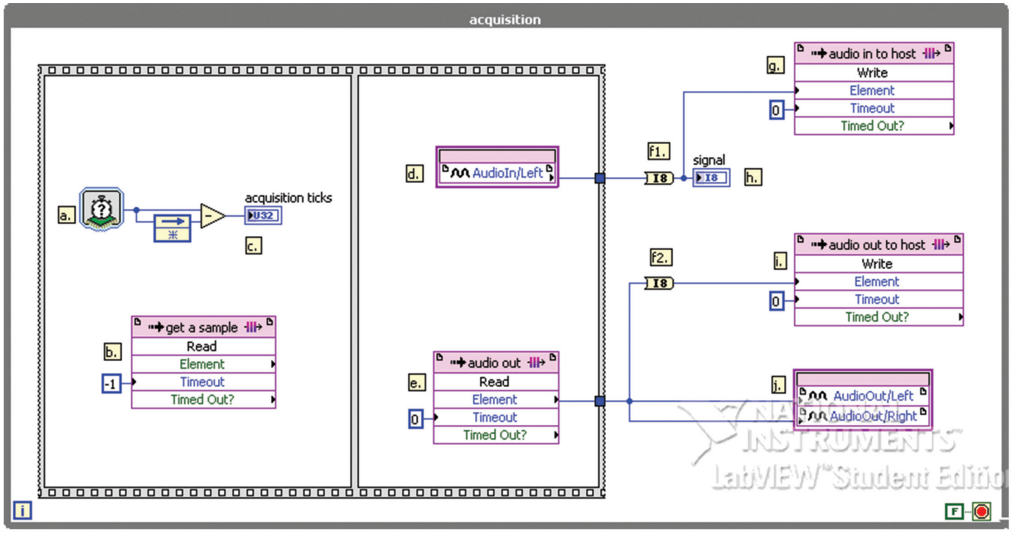
SHAW, A. C. *Real-Time Systems and Software*. 1. ed. Wiley, 2001. ISBN 9780471354901. Disponível em: <http://amazon.com/o/ASIN/0471354902/>. Citado na página 24.

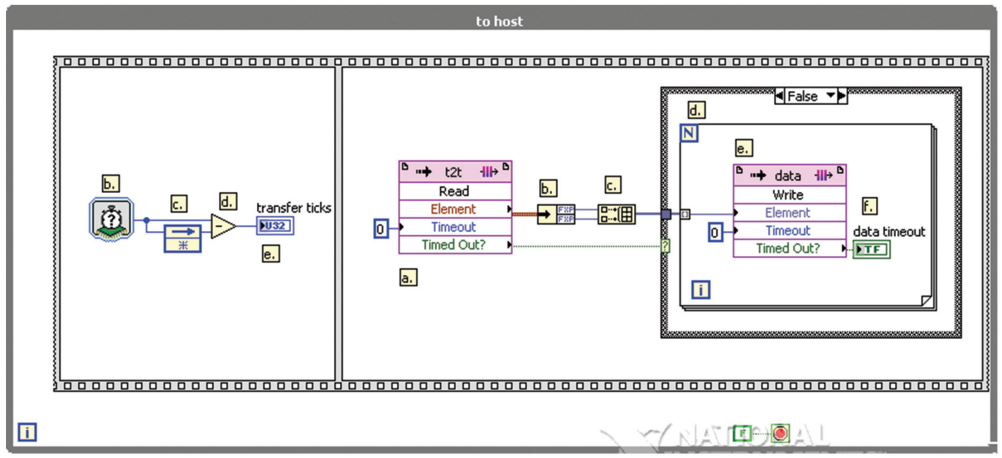
WIŚNIEWSKI, R. *Synthesis of Compositional Microprogram Control Units for Programmable Devices*. 2009. Disponível em: <http://zbc.uz.zgora.pl/Content/27955/Remigiusz_Wisniewski_Synthesis_of_CMCUs_for_Programmable_Devices.pdf>. Citado 2 vezes nas páginas 25 e 26.

Anexos

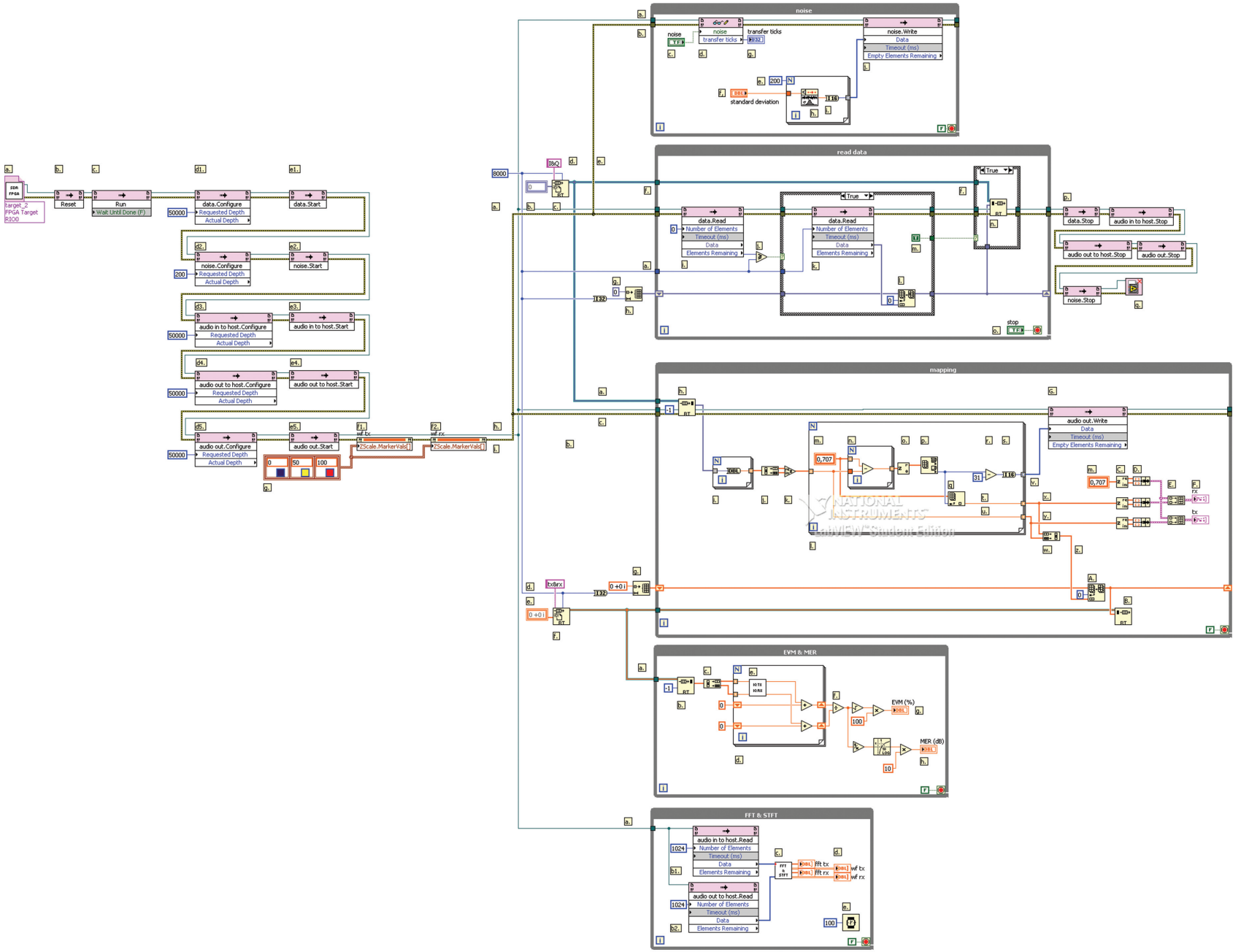




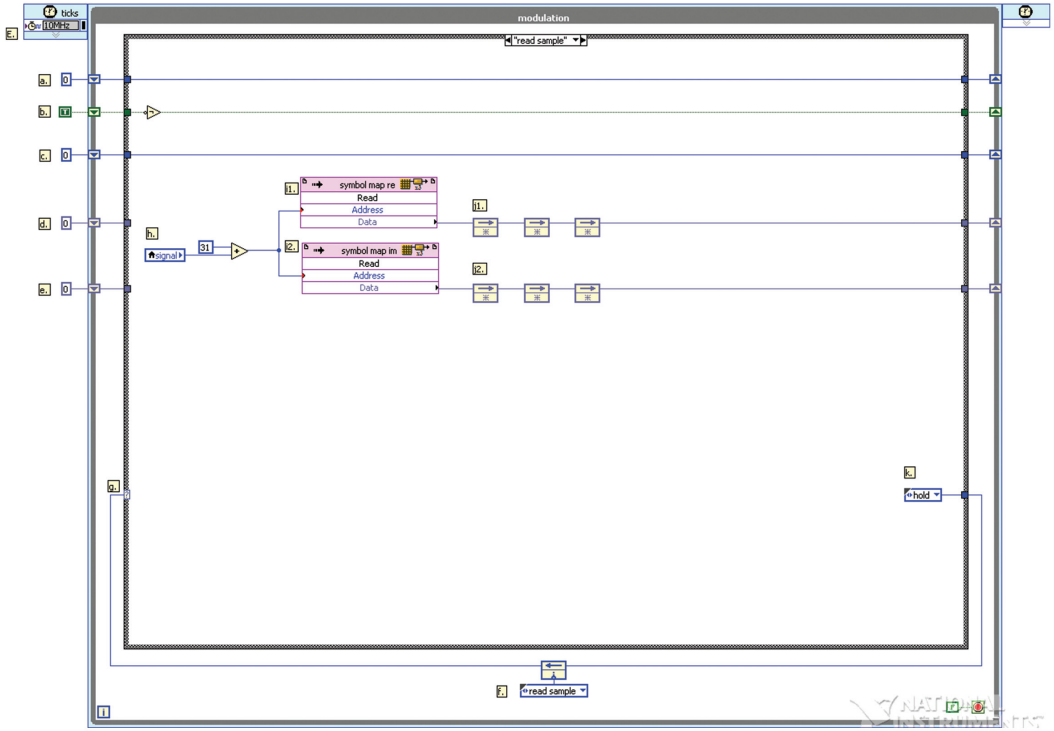


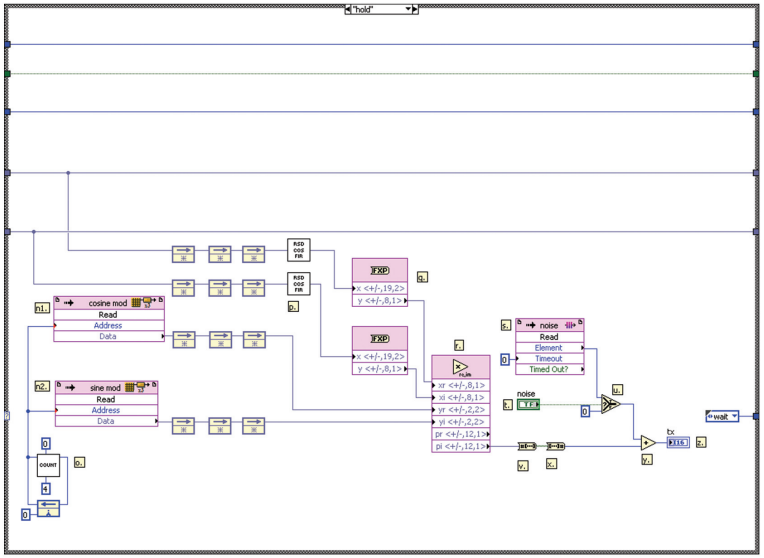


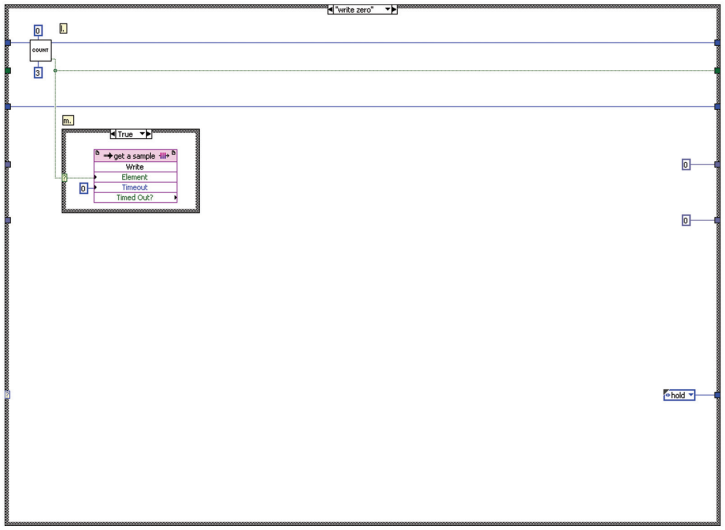
UFRGS UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL	*Loops* de aquisição, transmissão, recepção e de envio de dados ao *host*	
	Prancha: 1/5	Projeto de Diplomação - ENG04029
		Orientador: Prof. Dr. Hamilton Klimach
	Escala: 1/1	Aluno: Fábio Beck Wanderer

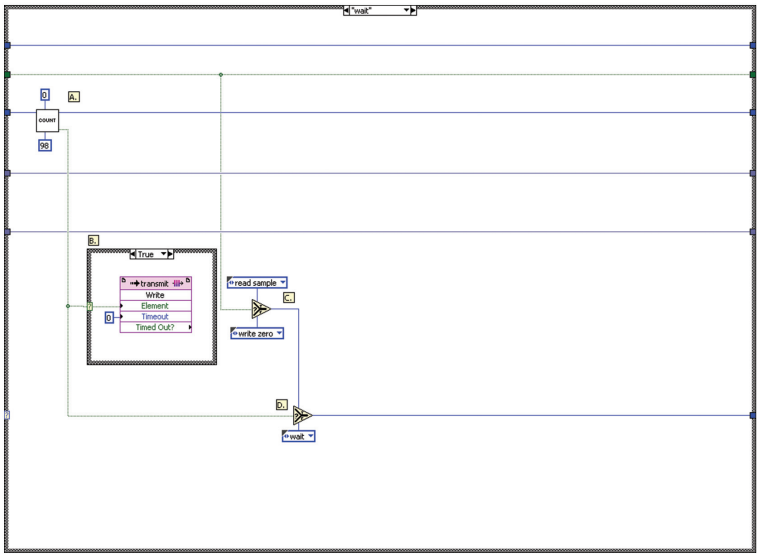


UFRGS UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL	*Block Diagram* da VI executada no *host*
Prancha: 2/5	Projeto de Diplomação - ENG04029
Escala: 1/1	Orientador: Prof. Dr. Hamilton Klimach
	Aluno: Fábio Beck Wanderer

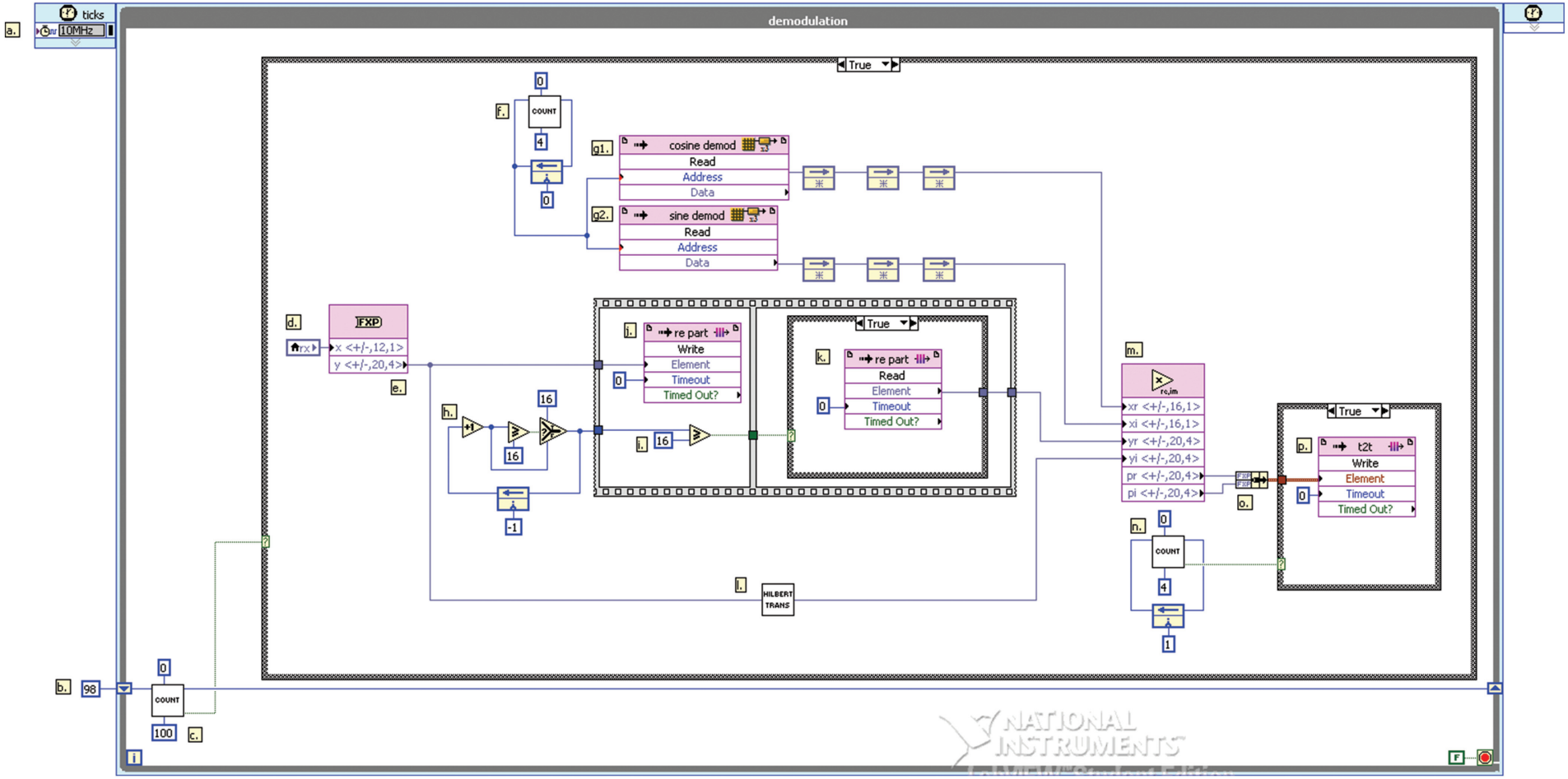




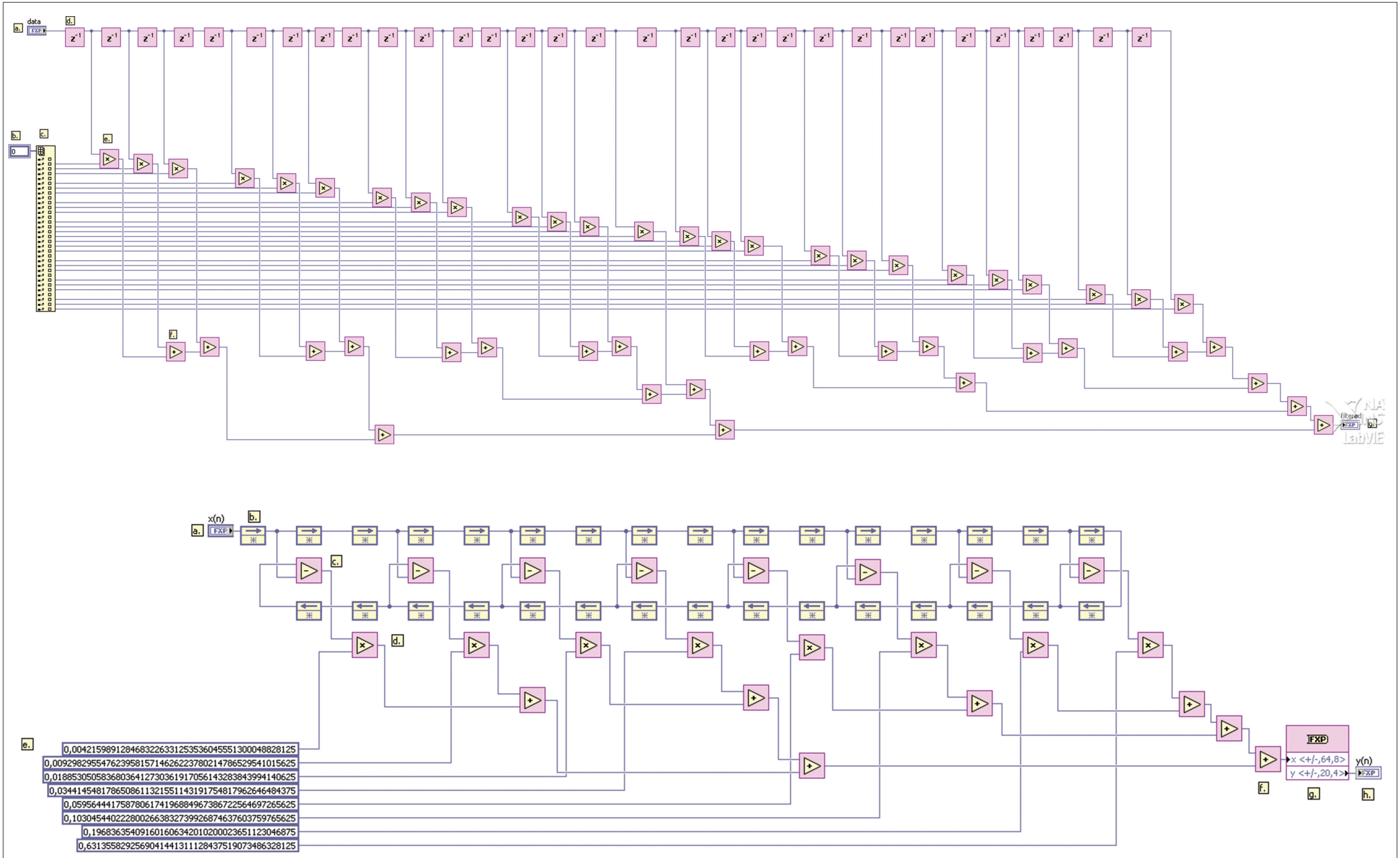




UFRGS UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL	*Loop* “modulation”	
	Prancha: 3/5	Projeto de Diplomação - ENG04029
		Orientador: Prof. Dr. Hamilton Klimach
	Escala: 1/1	Aluno: Fábio Beck Wanderer



UFRGS UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL	*Loop* “demodulation”	
	Prancha: 4/5	Projeto de Diplomação - ENG04029
		Orientador: Prof. Dr. Hamilton Klimach
	Escala: 1/1	Aluno: Fábio Beck Wanderer



UFRGS UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL	Diagrama de blocos das sub VI's "raised cosine fir" e "hilbert fir"	
	Prancha: 5/5	Projeto de Diplomação - ENG04029
		Orientador: Prof. Dr. Hamilton Klimach
	Escala: 1/1	Aluno: Fábio Beck Wanderer