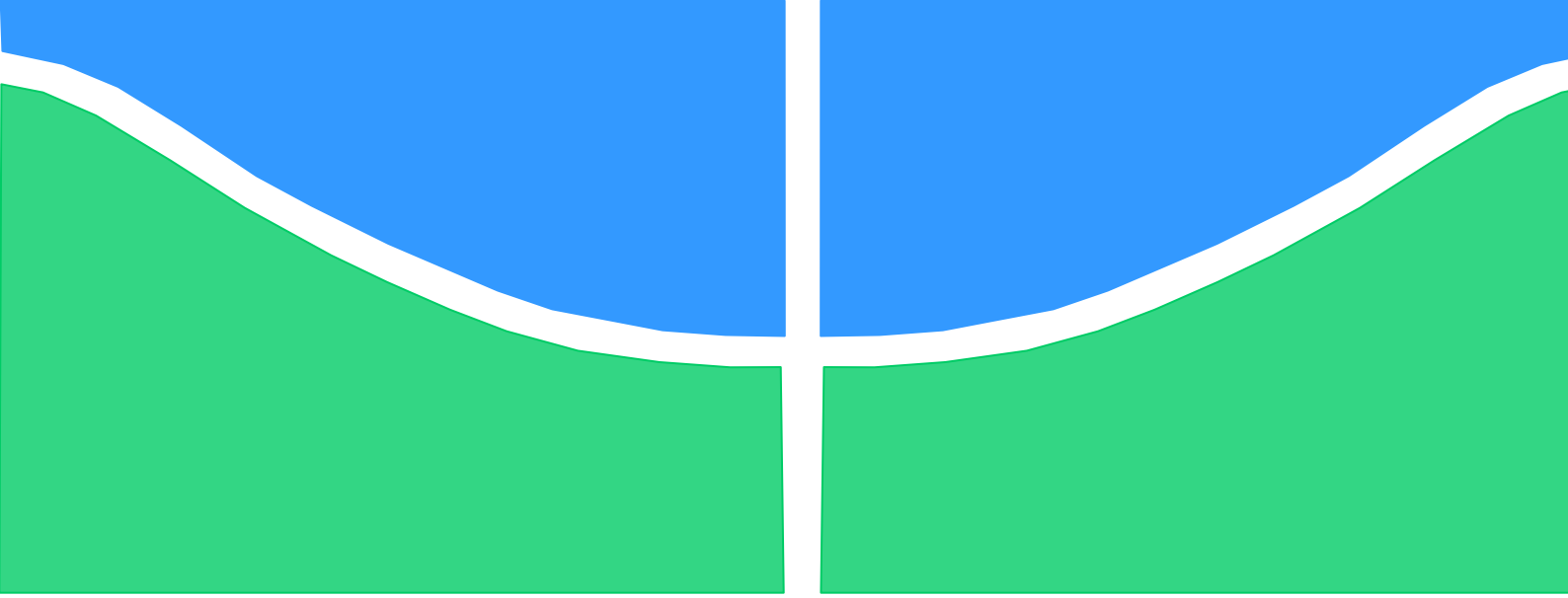

Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Curso de Engenharia Eletrônica

IMPLEMENTAÇÃO DE RÁDIO DEFINIDO POR SOFTWARE EM KIT DIDÁTICO FPGA

Autor: Pedro Henrique dos Santos Azevedo
Orientador: Leonardo Aguayo

Brasília, DF
2017

PEDRO HENRIQUE DOS SANTOS AZEVEDO

IMPLEMENTAÇÃO DE RÁDIO DEFINIDO POR SOFTWARE EM KIT DIDÁTICO FPGA

Monografia submetida ao curso de graduação em Engenharia Eletrônica da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia Eletrônica.

Orientador: Professor Leonardo Aguayo

Brasília, DF
2017

CIP – Catalogação Internacional da Publicação*

Santos Azevedo, Pedro Henrique.
Implementação de Rádio Definido por Software em Kit Didático FPGA / Pedro Henrique dos Santos Azevedo. Brasília: UnB, 2017. 103 p. : il. ;

Monografia (Graduação) – Universidade de Brasília Faculdade do Gama, Brasília, 2017. Orientação: Professor Leonardo Aguayo

1. FPGA. 2. MATLAB HDL-Coder. 3. VHDL. 4. Modulação Digital.
I. Aguayo, Leonardo. II. Implementação de Rádio Definido por Software em Kit Didático FPGA.

CDU Classificação

REGULAMENTO E NORMA PARA REDAÇÃO DE RELATÓRIOS DE PROJETOS DE GRADUAÇÃO FACULDADE DO GAMA - FGA

Pedro Henrique dos Santos Azevedo

Monografia submetida como requisito parcial para obtenção do Título de Bacharel em Engenharia Eletrônica da Faculdade UnB Gama - FGA, da Universidade de Brasília, no 2º semestre de 2017 apresentada e aprovada pela banca examinadora abaixo assinada:

Prof. Dr.: Leonardo Aguayo, UnB/ FGA
Orientador

Prof. Dr.: Daniel Mauricio Muñoz Arboleda, UnB/ FGA
Membro Convidado

Prof. Dr.: Wellington Avelino do Amaral, UnB/ FGA
Membro Convidado

Brasília, DF
2017

Dedicatória

Dedico este trabalho a minha família, meu pai José, minha mãe Leda e meu irmão Felipe.

Pedro Henrique dos Santos Azevedo

RESUMO

O presente trabalho apresenta o projeto de desenvolvimento e resultados de implementação de um RDS usando a placa de desenvolvimento FPGA Nexys 3. A implementação é construída por meio da integração da plataforma de simulação da ferramenta MATLAB, Simulink (em referência a funcionalidade HDL-Coder), com a placa de desenvolvimento Nexys 3. Com isso, o projeto é primeiramente implementado no ambiente de simulação MATLAB/Simulink, convertido para o nível VHDL usando Simulink HDL Coder e por fim aprimorado e sintetizado com o ISE Design Suite.

Os resultados ilustram o comportamento das modulações constituintes do RDS, responsáveis por modular/transmitir palavras digitais, símbolos, respeitando os parâmetros de configuração dos moduladores ASK, FSK, PSK e QAM.

Para a implementação do RDS, foi necessário integrar o sintetizador DDS junto aos blocos moduladores, matemáticos e auxiliares de modo a criar diferentes modos de executar distintas modulações digitais, de acordo com os requisitos determinados pelo projeto.

Palavras-chave: FPGA, MATLAB HDL-Coder, VHDL, Modulação Digital.

ABSTRACT

This work presents the development project and results of implementing an RDS using Nexys 3 board. The implementation is built by integrating of the simulation platform of the MATLAB tool, Simulink (in reference to HDL-Coder functionality) with the Nexys 3 development board. With this, the project is firstly implemented in the MATLAB/Simulink simulation environment, converted to VHDL level using Simulink HDL Coder and finally synthesized and improved with the ISE Design Suite.

The results illustrate the behavior of RDS constituent modulations, responsible for modulating/transmitting digital words, symbols, respecting the configuration parameters of ASK, FSK, PSK and QAM modulators.

For the implementation of RDS, it was necessary to integrate the DDS synthesizer next to the modulators, mathematicians and auxiliaries blocks in order to create different ways to execute different digital modulations, according to the requirements determined by the project.

Keywords: FPGA. MATLAB HDL-Coder. VHDL. Digital Communication.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

DAC – Digital to Analog Converter
DAS – Sintese Analógica Direta.
DDS – Sintese Digital Direta.
DSP – Digital Signal Processor.
FPGA – Field Programmable Gate Array
PLL – Phase Locked Loop.
RDS – Rádio Definido por Software.
VCO – Voltage Controlled Oscilator.
VHDL – VHSIC Hardware Description Language.
SIPO – Serial In Parallel Out.

LISTA DE SÍMBOLOS

Símbolo	Significado	Unidade
$A_c(t) / A_c$	Amplitude da portadora	volt (V)
$m(t)$	Mensagem a ser modulada/ transmitida	volt (V)
ω_c	Frequência da portadora	radiano por segundo (rad/s)
ω_i	Frequência instantânea	radiano por segundo (rad/s)
θ_c	Fase da portadora	radiano (rad)
$\theta_c(t)$	Fase instantânea	radiano (rad)
f_c	Frequência da portadora	hertz (Hz)
B	Largura de Banda	hertz (Hz)
$y(t)$	Sinal modulado	volt (V)
$\phi(t)$	Desvio de fase	radiano (rad)
Δf_i	Desvio da frequência instantânea	hertz (Hz)
k_p	Constante de desvio de fase	rad/V
k_f	Constante de desvio de frequência	Hz/V

SUMÁRIO

Capítulo 1

Introdução

1.1 Contexto

O Rádio Definido por Software (RDS), é uma tecnologia que emergiu nas últimas décadas, devido ao imenso cenário favorável à reconfigurabilidade de dispositivos para suprir uma alta demanda de acesso e busca de informações [17].

Assim, os anos de 1980 e 1990 presenciaram à explosão do RDS como um meio de comunicar todas as formas audíveis, visuais e informações geradas por máquinas sobre extensas distâncias [3]. Nos últimos anos, os sistemas de rádio analógico estão sendo substituídos por sistemas de rádio digital para várias aplicações em espaços militares, civis e comerciais. Além disso, a indústria de comunicações wireless vem enfrentando problemas devido a constante evolução dos padrões de protocolo da camada de enlace (2G, 3G, 4G). A existência de tecnologias de rede incompatíveis em diferentes países que inibem a implantação de roaming global e o enfrentamento de problemas na implantação de novos serviços/funcionalidades ocorre devido à presença de tecnologia obsoleta herdada por alguns consumidores [4].

O RDS compreende tanto hardware como software, usando a capacidade reprogramável do *Field Programmable Gate Array* (FPGA) ou do *Digital Signal Processor* (DSP) para construir uma arquitetura aberta com implementação de software de frequências de rádio como modulação/demodulação, codificação/decodificação, etc [5].

Além da reconfigurabilidade, outra grande vantagem do uso do RDS é o barateamento do custo dos dispositivos, de forma que os equipamentos teriam uma produção mais padronizada em hardware, sendo que as funcionalidades de cada aparelho seriam definidas por um software instalado posteriormente [19].

Assim, preveem-se boas expectativas de crescimento do uso do RDS, qual apresenta um aumento na capacidade do sistema e, por conseguinte propicia um aumento de serviços aos usuários. Crescimento esse, ilustrado na Figura 1, qual constrói uma analogia com as gerações de telefonia móvel [17].

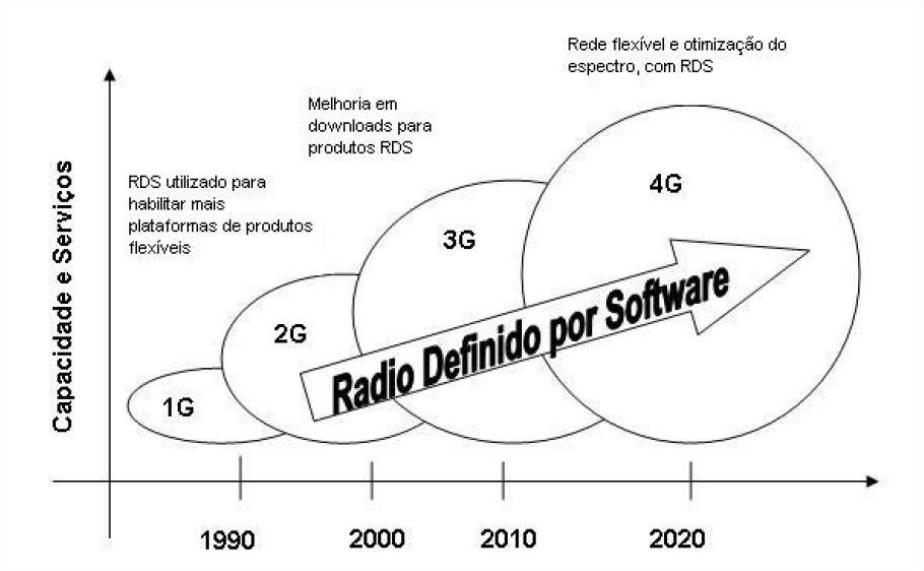

Figura 1. Usos do RDS [17].

1.2 Definição do problema.

A lacuna entre os padrões da camada de enlace e a problemática de roaming global, traz à tona a conveniência da construção de um rádio digital que mude suas funcionalidades de modulação (analógicas e/ou digitais) usando o controle de software, visando à otimização e aplicação de uma comunicação digital que atenda ao requisito de faixa de operação de frequências na escala de Mega Hertz (MHz).

1.3 Objetivo Geral

O objetivo deste trabalho consiste em aplicar conceitos de Engenharia Eletrônica, na área de Telecomunicações, e desenvolver um RDS que implemente diferentes formas de modulação digitais, entregando alta performance junto ao controle de parâmetros de variação.

1.4 Objetivos Específicos

Os objetivos específicos são:

- Projetar e implementar um RDS utilizando a linguagem VHDL;
- Projetar e implementar modulação digital 8-ASK;
- Projetar e implementar modulação digital 8-FSK;

- Projetar e implementar modulação digital 8-PSK;
- Projetar e implementar modulação digital 16-QAM;

1.5 Organização do Trabalho

A dissertação está estruturada da seguinte maneira, no Capítulo 2 é feita uma definição de todos os conceitos e ferramentas necessárias para a construção do RDS, trazendo a definição de modulações analógicas e digitais, DDS, conversores, FPGA, VHDL e suas respectivas aplicações.

O Capítulo 3 aborda aspectos de descrição de como o projeto foi realizado, descrevendo as tarefas a serem executadas junto com as ferramentas utilizadas.

O Capítulo 4 traz, por conseguinte, toda a parte de projeto e implementação do RDS, descrevendo todo o processo de construção e integração dos blocos constituintes do RDS. No Capítulo 5 são apresentados os resultados referentes ao projeto, elencando uma discussão e validação em cima de gráficos e tabelas ilustradas. Por fim, no Capitulo 7, encontram-se as considerações finais sobre os estudos realizados e o futuro desse trabalho.

Capítulo 2

Conceitos Básicos de Modulações Analógicas e Digitais

Modulação faz referência ao processo qual se desloca o sinal de mensagem para uma banda especifica de frequências, de modo que essa mensagem possa ser transmitida em um dado canal/meio de propagação. Em um sinal analógico pode-se variar sua amplitude, frequência e fase, de maneira com que cada modo de variação caracteriza uma diferente forma de modular um sinal, sendo eles [6]:

- Modulação em Amplitude (AM);
- Modulação em Frequência (FM);
- Modulação em Fase (PM);

2.1 Modulações

2.1.1 Modulação em Amplitude (AM)

Modulação em Amplitude é uma metodologia de modulação em que a amplitude $A_c(t)$ é um sinal senoidal, denominado portadora, varia em função do sinal de banda base, ou seja, da mensagem a ser transmitida (sinal modulante), representada por $m(t)$. Pelo fato de apresentar variação na componente de amplitude, a frequência ω_c e a fase θ_c da portadora permanecem constantes, sendo que a fase é igualada a zero. Considerando que a amplitude da portadora $A_c(t)$ seja diretamente proporcional ao sinal modulante $m(t)$, o sinal modulado será $m(t)\cos(\omega_c t)$, como ilustrado na Figura 2, esse tipo de modulação simplesmente desloca o espectro de $m(t)$ para frequência da portadora, f_c,ilustrado na Figura 3 [6].

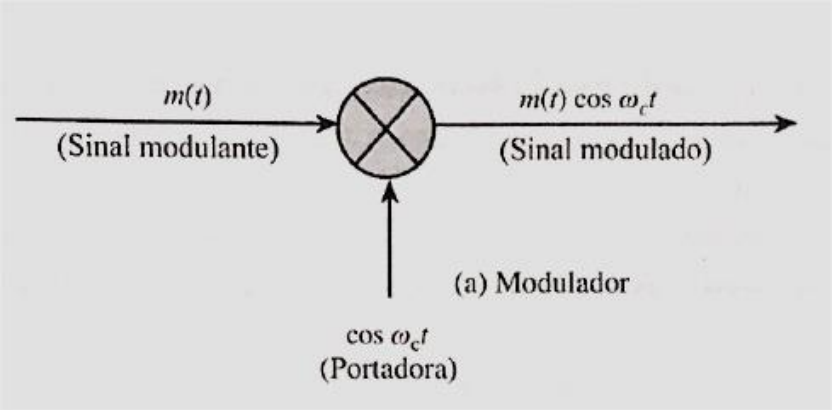

Figura 2. Diagrama Modulação AM [6].

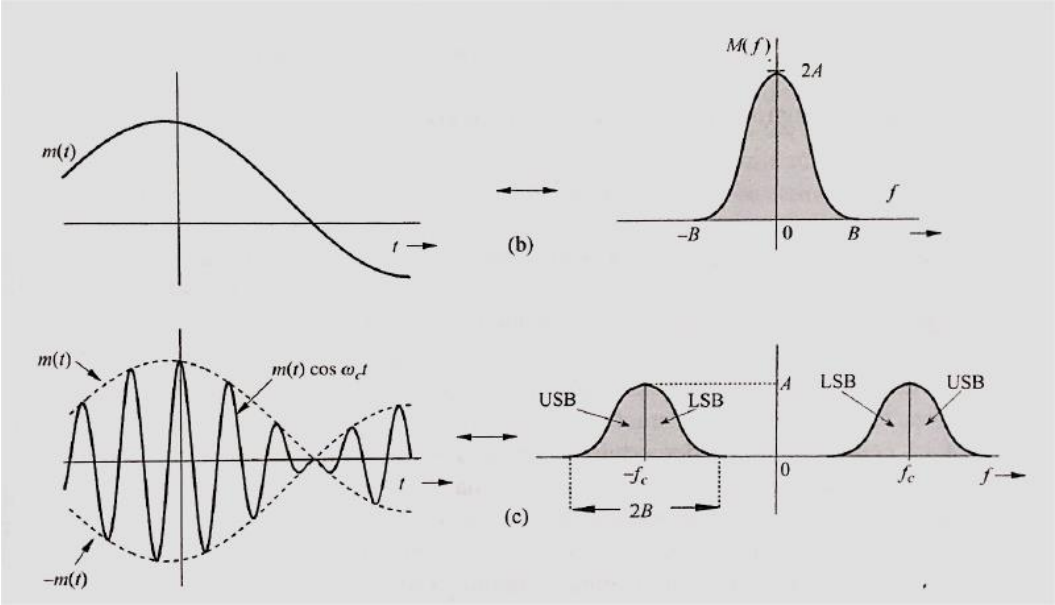

Figura 3. Espectro do sinal modulante e do sinal modulado [6].

Como observado na Figura 3, o processo de modulação desloca o espectro do sinal modulante para a esquerda e para a direita por f_c. Nota-se, portanto que, se a largura de banda de $m(t)$ for B, o sinal modulado apresentará largura de banda $2B$. Ressaltando que a relação entre f_c e B é bastante relevante, de modo que, $f_c \geq B$ evita a sobreposição dos espectros modulados e centrados em $-f_c$ e $+f_c$, onde de maneira contrária se $f_c < B$, as duas componentes do espectro da mensagem se sobreporiam e a informação $m(t)$ seria perdida, impossibilitando a sua recuperação [6].

2.1.2 Modulação ASK

Se caracteriza pela forma digital de se implementar uma modulação AM, construída em cima do deslocamento de amplitude, pode ser estabelecida por meio do método ON-OFF, qual consiste em representar os bits 0 e 1 de um sinal digital pela ausência desse, ou seja, tensão nula ou pela presença do sinal da portadora, respectivamente. Na Figura 4 apresenta-se o sinal modulado $y(t)$ e o sinal modulador/mensagem $m(t)$. A fase e a frequência da portadora não se alteram [6].

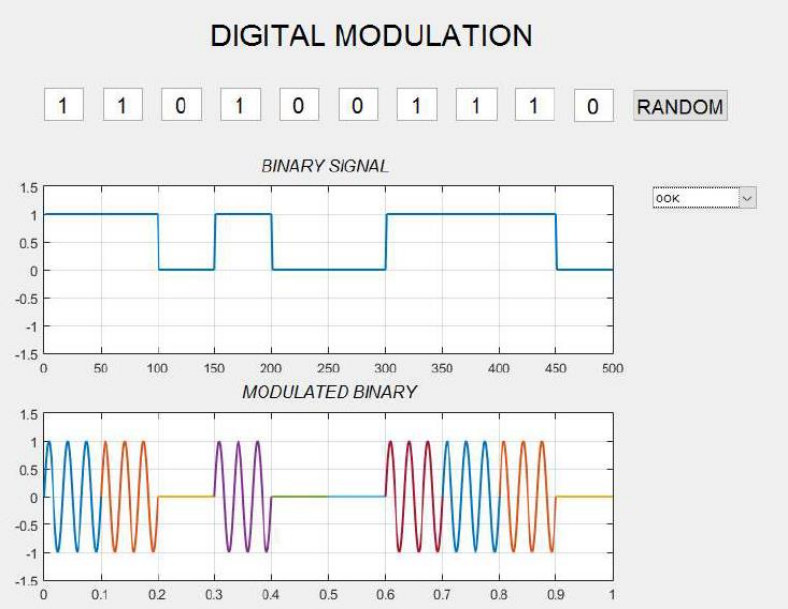

Figura 4. Modulação ASK [9].

2.1.3 Modulação em ângulo

Modulação angular consiste em um método de comunicação em que a informação a ser transmitida modula o ângulo (frequência ou fase) da portadora, cuja amplitude é mantida constante. De modo geral, o sinal modulado é representado por:

$$x_c(t) = A_c \cos(\theta_c(t)) \tag{1}$$

onde A_c é a amplitude da portadora e θ_c (t) é a fase instantânea [6].

Define-se a fase instantânea por, $\theta_c(t) = 2\pi f_c t + \phi(t)$ em que f_c é a frequência de portadora e $\phi(t)$ é o desvio de fase.

O desvio de fase $\phi(t)$ da portadora varia com o sinal modulante $m(t)$. Dependendo da relação entre $\phi(t)$ e $m(t)$, há duas formas de programar a modulação angular, Modulação em Fase (PM) e a Modulação em Frequência (FM). Em PM, o desvio de fase instantâneo da portadora é proporcional ao sinal modulante, isto é [6]:

$$\phi(t) = k_p m(t) \quad (2),$$

onde k_p é a constante de desvio de fase (rad/V).

A frequência instantânea ω_i é definida por, $\omega_i = \frac{d\theta_c}{dt} = \omega_c + \frac{d\phi}{dt}$, já o desvio da frequência instantânea (em Hertz) do sinal modulado é definido como, $\Delta f_i = \frac{1}{2\pi}\frac{d\phi}{dt}$. Assim, em FM, o desvio de frequência da portadora é proporcional ao sinal modulante, definido por [6]:

$$\phi(t) = 2\pi k_f \int_{t_0}^{t} m(\lambda)d\lambda + \phi(t_0) \quad (3),$$

de modo que, k_f é a constante de desvio de frequência (Hz/V) e $\phi(t_0)$ é o ângulo inicial em $t = t_0$, qual é igualado à zero. Por fim, combinando (2) e (3) com (1), obtêm-se as seguintes equações características de cada modulação [6]:

PM: $$x_c(t) = A_c \cos\left[\omega_c t + k_p m(t)\right] \quad (4)$$

FM: $$x_c(t) = A_c \cos\left[\omega_c t + 2\pi k_f \int_{-\infty}^{t} m(\lambda)d\lambda\right] \quad (5)$$

As equações (4) e (5) revelam que os sinais modulados em fase e frequência são similares em suas composições, onde ambos mostram que tanto em PM como em FM, o ângulo de uma portadora varia proporcionalmente em alguma medida de $m(t)$. Em PM, o ângulo é diretamente proporcional a $m(t)$, enquanto em FM, é proporcional a integral de $m(t)$. A Figura 5 ilustra a equivalência matemática das modulações, em que um modulador em frequência pode ser utilizado diretamente para gerar um sinal FM ou a mensagem modulante $m(t)$ pode ser processada por um filtro (diferenciador) para gerar um sinal PM. Do mesmo modo que se pode utilizar um modulador em fase para gerar um sinal PM ou processar a mensagem modulante *m(t)* através de um filtro (integrador) para gerar um sinal FM [6].

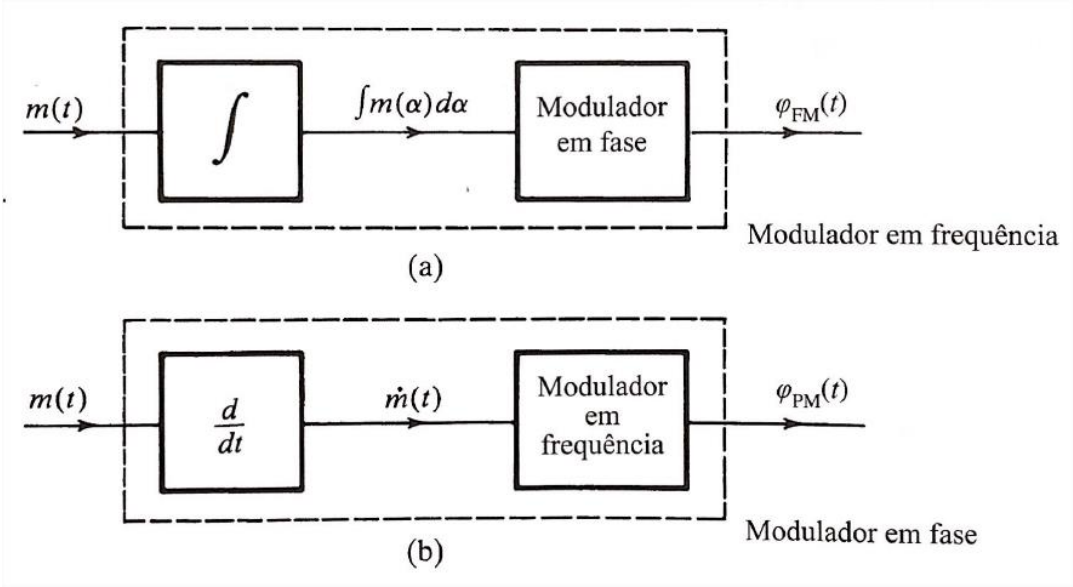

Figura 5. Equivalência das modulações em Fase e Frequência [6].

Por meio das Figuras 6 e 7, a seguir, ilustram-se sinais modulados em ângulo, tanto PM quanto FM, respectivamente.

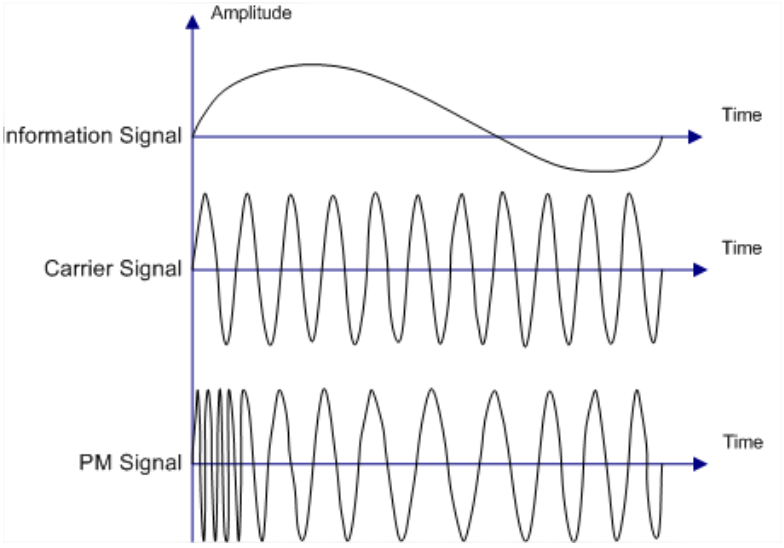

Figura 6. Ilustração da Modulação em Fase (PM) [7].

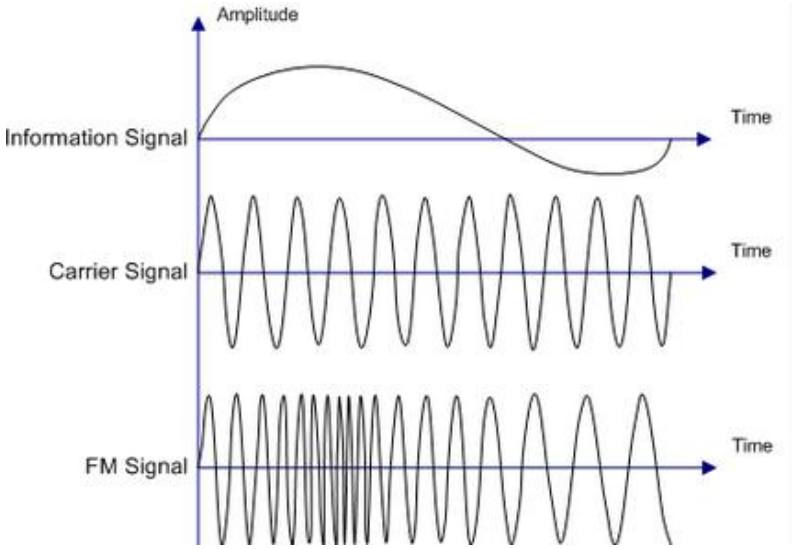

Figura 7. Ilustração da Modulação em Frequência (FM) [7].

2.1.4 Modulação FSK

A forma digital de implementar a modulação FM, é dada pela modulação FSK, onde os bits 0 e 1 são associados a diferentes valores de frequência, ou seja, para transmitir o bit 1, a portadora assume a frequência *f1*, e para transmitir o bit 0, a portadora assume a frequência *f2*. Como característica do comportamento da modulação, a amplitude permanece constante. A Figura 8 ilustra o sinal modulado $y(t)$, o sinal modulador/mensagem $m(t)$ e as respectivas frequências *f1* e *f2* [6].

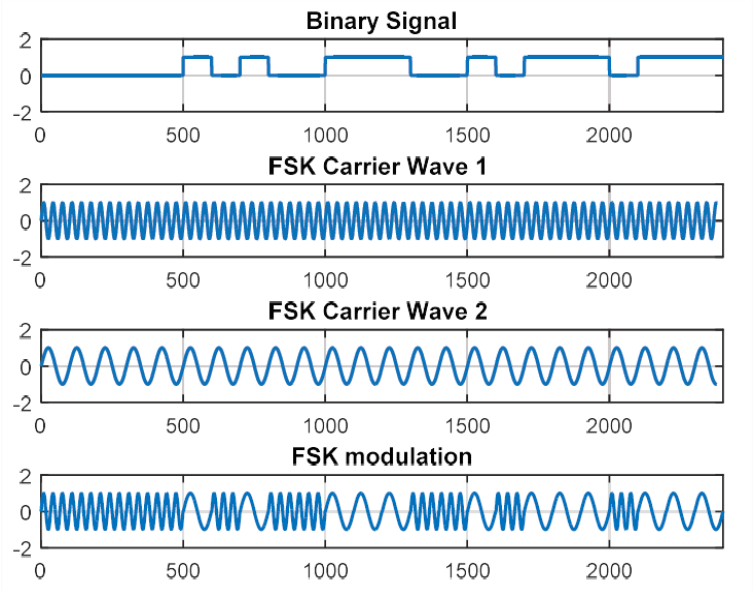

Figura 8. Modulação FSK [9].

2.1.5 Modulação PSK

A forma digital de programar a modulação PM, é dada pela modulação PSK, onde os bits 0 e 1 são associados a mudanças na fase da portadora e a frequência permanece constante. Um método muito implementado consiste na inversão de 180º a cada mudança de bit. Na Figura 9 ilustra-se a aplicação de tal modulação, observando o sinal modulado $y(t)$ e o sinal modulador/mensagem $m(t)$ [6].

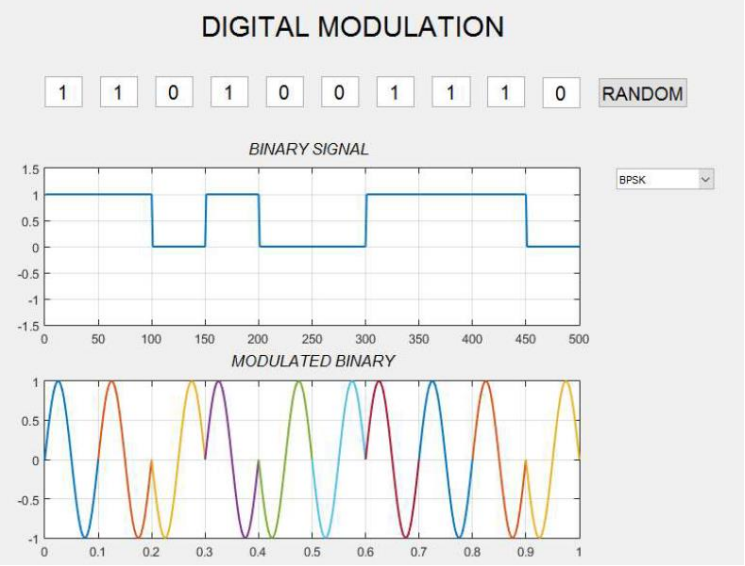

Figura 9. Modulação PSK [9].

2.1.6 Modulação QPSK

A modulação QPSK é uma técnica de modulação derivada do PSK, porém neste caso são utilizados parâmetros de fase e quadratura da onda portadora para modular o sinal de informação. Como são utilizados dois parâmetros de variação, permite-se que sejam transmitidos mais bits por símbolo (conjunto de bits) [6].

Dado um caso, qual queira transmitir 2 bits por símbolo, tem-se 4 tipos de símbolos possíveis, onde a portadora pode assumir 4 valores de fase diferentes, como pode ser observado na Tabela 1 [6].

Símbolo	Mudança de Fase
00	+0º
01	+90º
10	+180º
11	+270º

Tabela 1. Mudança de fase x Símbolo.

De modo que, a diferença entre a modulação PSK e a QPSK, pode ser observado pela Figura 10, podendo notar diferentes mudanças de fase para cada símbolo.

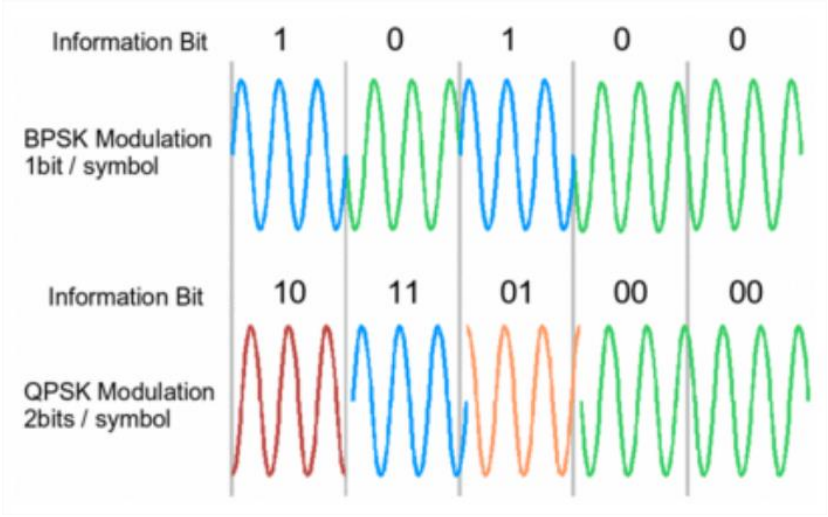

Figura 10. Modulação QPSK [9].

2.1.7 Modulação QAM

A modulação QAM é uma técnica de modulação derivada do ASK, porém neste caso os símbolos são mapeados em um diagrama de dispersão bidimensional no plano complexo em instantes de amostragem de símbolo, em que cada instante o símbolo ocupa apenas uma posição no diagrama, representando fase e quadratura. A Figura 11 indica um diagrama (constelação) de representação de símbolos, onde no caso apresenta 16 símbolos, o fator pode variar de acordo com a quantidade de símbolos a serem representados [6].

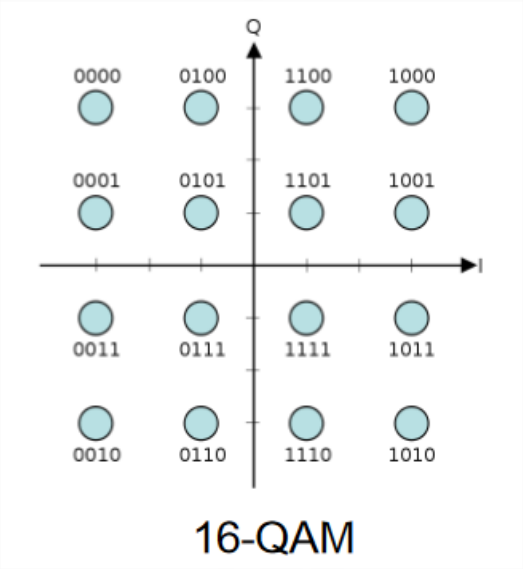

Figura 11. Constelação 16-QAM [11].

Os pontos no diagrama são nomeados pontos de constelação, em que em cada instante do tempo o símbolo é representado por módulo e fase diferentes. Em telecomunicações digitais os dados são geralmente binários e o número de símbolos é dado por uma potência de 2. Na Figura 12 ilustra-se o comportamento de uma modulação 16-QAM.

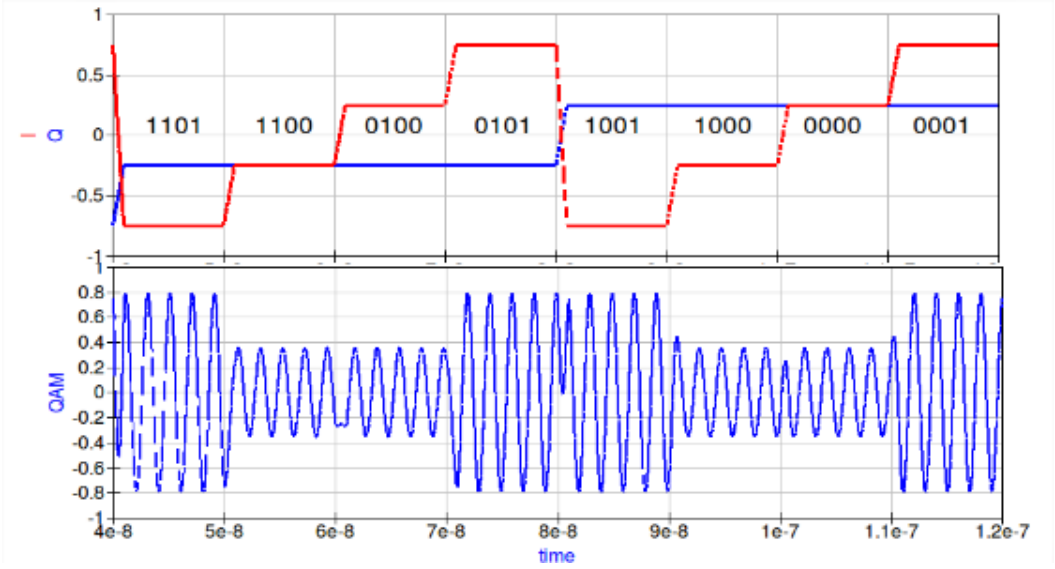

Figura 12. Modulação 16-QAM [11].

2.2 Síntese Digital Direta

Em relação aos conceitos de modulação descritos acima, surgem maneiras de implementação totalmente digitais, as quais fazem necessário a geração de sinais digitais que se comportem como senos e cossenos variáveis, representando assim a portadora, qual permite a implementação dos diferentes métodos.

Síntese Digital Direta, DDS (*Direct Digital Synthesis*) é uma técnica de geração de sinais digitais com determinadas fases e frequências precisamente controladas e referenciadas a uma frequência de *clock* padrão. A Figura 13 ilustra a arquitetura padrão de um DDS [8].

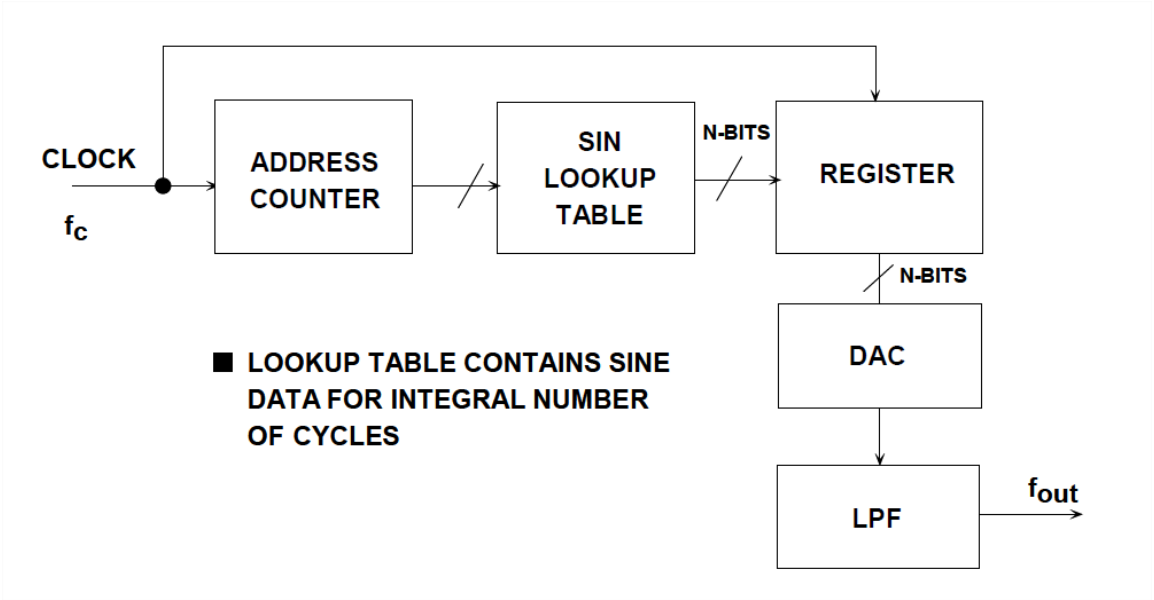

Figura 13. Arquitetura DDS [12].

A arquitetura ilustrada acima faz uso de valores retirados de uma função periódica armazenada em uma memória de leitura programável (PROM), ao qual a cada ciclo de *clock*, o contador de endereços (ADDRESS COUNTER) passa por um valor da função periódica armazenada na memória, que por sua vez é emitido através de um DAC que gera um sinal analógico. A frequência de saída do DDS depende da frequência de referência do *clock* e do tamanho do valor de incremento definido dentro do contador de endereços, configurado na PROM. O contador de endereços e o registrador funcionam como um acumulador qual aumentam o valor

da saída de acordo com a variável de incremento do contador a cada ciclo de *clock* [8].

A saída do acumulador assume a forma de um endereço usado pela memória, a qual contém as amostras da forma de onda. O número de ciclos de *clock* necessários para percorrer toda a memória define o período da forma de onda. A memória possui a representação digital da forma de onda desejada, dada por palavras digitais que definem a amplitude da onda como uma função da fase. Cada posição gerada pelo contador de endereços representa um valor da fase da onda, onde caso a memória possua 512 endereços, a posição 0 corresponde a fase 0º enquanto a posição 511 corresponde a fase (511/512)*360º da onda [8].

O contador de endereços lê sequencialmente a tabela de valores digitais da memória e os passa para um DAC, o qual gera a forma de onda de saída. Os DACs alteram cada uma dessas palavras digitais para uma tensão analógica, produzindo assim uma forma de onda determinada pela configuração do usuário [8].

2.2.1 Síntese Analógica de Sinais

Técnicas comuns de geração de sinais incluem a Síntese Analógica Direta (DAS) e o método PLL. Com isso, DAS envolve a geração de várias frequências através da combinação de diferentes cristais ou agrupando os seus harmônicos. Apresentando assim limitação em seu tamanho, custo e consumo de energia, dificultando a sua aplicação em objetos portáteis. Porém, ainda são amplamente utilizados em aplicações de testes de alta velocidade, como imagens médicas e radares em geral [8].

Já o PLL é um mecanismo de feedback que rastreia uma frequência de referência, composto pelo acoplamento do VCO, com um detector de fase e um filtro de loop. Sintetizadores PLL’s podem oferecer uma resolução de frequência fina a um custo relativamente baixo, porém não atingem níveis tão baixos de ruídos como os apresentados pelos DAS’s, tendo tempo de comutação de frequências reduzidos devido ao atraso de ajuste do filtro de loop [8].

Grande parte dos problemas/falhas descritos pelos sintetizadores descritos são resolvidos pelo DDS, que é superior em termos de precisão, facilidade de implementação e flexibilidade, levando-o a um uso generalizado em sistemas de comunicações digitais [8].

De modo que, com as características de cada um dos sintetizadores descritos, constrói-se a Tabela 2, referente a vantagens e desvantagens do DDS.

VANTAGENS	DESVANTAGENS
Precisão ao definir a frequência, podendo alcançar altas resoluções;	Tamanho do equipamento: técnicas para eliminação de impurezas do sinal podem aumentar o tamanho do equipamento;
Flexibilidade de mudanças em resolução e largura de banda;	Largura de banda: frequência de saída limitada pelo teorema de Nyquist;
Facilidade de implementação, qual implementações em alta escala são baratas e altamente disponíveis;	Pureza espectral: perda de sinais quando há truncamento de fase;
Comutação de Frequências: alta velocidade e suavidade de transição;	
Tamanho do equipamento: pode ser implementado em uma fração do tamanho de um sintetizador analógico similar;	
Largura de Banda: pode variar mudando a velocidade do clock, e pode ser aumentada usando um DSS acoplado ao PLL;	
Pureza espectral: alta qualidade dos sinais quando o tamanho do acumulador é um múltiplo do tamanho passo e não truncamento de fase.	

Tabela 2. DDS: Vantagens x Desvantagens [8].

2.3 Conversores Digital/Analógicos (D/A)

Realizam a conversão de números digitais para valores analógicos. Em sua maioria são circuitos assíncronos e que se utiliza de analógicos para realizar a conversão. Suas aplicações são amplas e seu uso é muito comum quando envolve a integração de sistemas digitais e analógicos em um mesmo circuito, o que é facilmente observado na estrutura do RDS. A conversão de valores binários para valores analógicos, ambos em tensão, é realizada por circuitos lineares capazes de somar tensões com pesos proporcionais aos pesos dos bits de resolução que individualmente produzem cada tensão [20].

2.4 FPGA

Field Programmable Gate Arrays (FPGA), são dispositivos semicondutores baseados em uma matriz de blocos lógicos, composto por processadores, interfaces, controladores, decodificadores, portas lógicas e *flip-flops*, todos configuráveis e conectados através de interconexões programáveis. Podem ser programados e reprogramados dados diferentes requisitos de aplicação ou funcionalidade de um determinado projeto. Diferem-se dos ASICs (*Application Specific Integrated Circuits*), da maneira com que esses são fabricados de forma personalizada para atender tarefas especificas de um único projeto, não permitindo sua reprogramação [14].

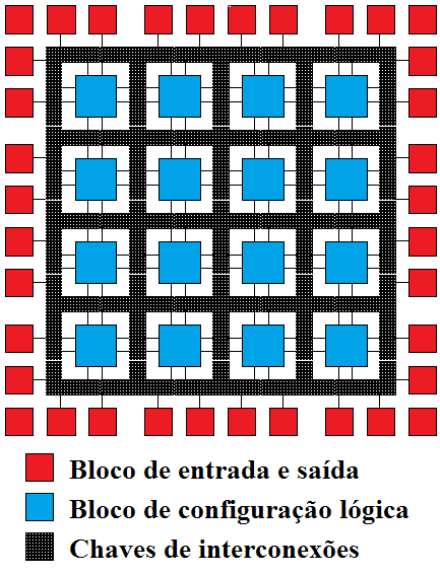

Figura 14. Arquitetura simplificada de um FPGA [25].

As linguagens utilizadas para a programação das FPGAs são as linguagens de descrição de Hardware, como o VHDL. O programa é executado por um processo de síntese que gera uma *netlist* a partir do código desenvolvido, sendo esse um arquivo binário que é carregado ao FPGA, responsável por configurar os blocos, as portas lógicas programáveis e suas respectivas chaves de conexão [16].

2.4.1 Aplicações

Devido ao seu comportamento programável, as FPGAs são amplamente adotadas em diferentes mercados. Sendo alguns deles [15]:

- Transmissão – adapta-se a mudanças de requisitos rapidamente e aumenta o ciclo de vida do produto com plataformas de projeto segmentado e soluções de transmissão de alto nível.
- Comunicações *Wired* – soluções de ponta a ponta para processamento de pacotes *Reprogrammable Networking Linecard, Framer/MAC e backends* em série.
- Comunicações *Wireless* – RF, banda base, conectividade, transporte e soluções de rede para equipamentos sem fio, atendendo a padrões como WCDMA, HDSOA e WiMAX.

Em meio a uma integração dos mercados descritos junto a reconfigurabilidade em hardware, obtida por meio da arquitetura do dispositivo, esta é um grande atrativo para o desenvolvimento do RDS, devido as suas características de projeto variáveis. Tornando a implementação do RDS em FPGA válida, devido ao seu alto poder de processamento e velocidade em hardware, junto ao alto desempenho e flexibilidade do sistema [8].

2.4.2 Análise comparativa com Sistemas DSPs

DPSs são microprocessadores com características peculiares, os quais podem ser programados e projetados para processar sinais digitais em tempo real, superando a velocidade de microprocessadores comuns. Possuem arquitetura específica e executam instruções de forma a realizar operações matriciais complicadas, como convoluções e FFTs. A programação de DSPs têm por base as

linguagens de alto nível como C ou C++, mas podem ser desenvolvidas em baixo nível usando Assembly [17].

Assim, se entrega diversas otimizações, visando 3 pilares principais, capacidade de controle (adicionar maior recurso de memória e processamento, em aplicações que requerem um maior controle das interfaces físicas), consumo de energia (dispositivos portáteis que consumem menos energia) e o range de performance [17].

Assim sendo, o DSP tem alta reconfigurabilidade, dada pelo upload de diferentes programas, do mesmo modo como a FPGA e apresenta baixo custo de aquisição comparado a outros sistemas, devido a sua popularidade em diferentes aplicações [17].

Consequentemente, comparando os recursos de reconfigurabilidade e desempenho dos dispositivos, constrói-se a Tabela 3.

Característica	**DSP**	**FPGA**
Frequência de operação (MHz)	100-600	100-300
Consumo de energia	Muito alta	Alto
Execução paralela	Serial	Máxima (flexível)
Complexidade de projeto	Programas Complexos	Muito alta
Tamanho	Moderado	Muito grande
Evolução	Alta	Alta
Customização	Muito fácil	Fácil
Verificação de projeto	Moderada	Moderada
Ferramentas de projeto	Bom	Muito bom

Tabela 3. Comparação DSP x FPGA [14, 15 e 17].

Analisando os dados comparativos, pode-se aferir que dispositivos DSPs possuem melhor tempo de reprogramação, flexíveis com alto poder de fácil customização, porém essa maior flexibilidade está intimamente ligada a baixo desempenho. Por outro lado, dispositivos FPGAs apresentam um tempo de reprogramação maior, entretanto com um hardware configurável, destacando-se também com um melhor desempenho, justificando assim o seu uso no projeto [17].

2.4.3 Análise comparativa com *Raspberry Pi 3*

A placa *Raspberry Pi 3*, é uma placa microcomputador de alta performance e processamento, qual permite diversas aplicações embarcadas em *Linux* ou *Windows* desenvolvidas em diferentes linguagens, como *Phyton*, C e C++ [26].

Tendo essa, as seguintes especificações, CPU AMRv8 *quad-core* de 64 bits de 1.2 Giga Hertz (GHz), LAN *Wireless* 802.11n, *Bluetooth* 4.1, *Bluetooth Low Energy*, 1 GB de RAM (expansível), 4 portas USB, 40 pinos I/O, porta HDMI, conexão com a rede, entrada de áudio/vídeo de de 3.5 mm, interface de câmera (CSI), interface de exibição (DSI), *slot* para cartão micro SD, núcleos de gráficos 3D *VideoCore* IV [26].

Analisando as especificações acima, observa-se que a placa em questão, não atende ao requisito de faixa de operação de frequências, funcionando em 1.2 Giga Hertz, estando assim longe da condição de operação do RDS, definido em Mega Hertz (MHz), assim, mais uma vez justifica-se o uso do kit FPGA.

2.5 VHDL

O VHDL é uma linguagem empregada na descrição de circuitos digitais. Inicialmente apenas uma ferramenta para documentação e modelamento de dispositivos de hardware, que hoje é largamente utilizada em sua implementação, graças ao surgimento de programas de síntese que permitem a criação de estruturas de circuitos lógicos diretamente das descrições de funcionamento em VHDL [21].

A estrutura básica de descrição de uma unidade em VHDL é relativamente simples, composta por duas partes. A primeira, denominada *entity declaration*, é a definição estrutural do circuito, nesta etapa são apenas designados os sinais de entrada e saída, sem que se definam relações entre os mesmos e nenhum detalhamento acerca do funcionamento do dispositivo. A segunda parte da estrutura é denominada *architecture definition*, onde definem se todas as funções, constantes, procedimentos, sinais, de modo a implementar um comportamento particular da unidade introduzida na primeira parte [21].

Capítulo 3

Metodologia de Projeto

3.1 Descrição

A implementação do RDS, pode ser desmembrada na resolução das seguintes tarefas:

- Projeto *Simulink* – construção inicial e simulação do DDS (*Wave Generator*).
- Projeto *HDL Coder* – transição do projeto *Simulink* para nível VHDL.
- Projeto *ISE* – codificação VHDL (FPGA).
- Simulação *VIVADO* – testes de simulação e validação.

Espera-se que ao término da resolução das atividades listadas, o RDS esteja implementado de modo a atender os requisitos de projeto.

3.2 Fluxograma

Tomando como base as atividades acima, constrói-se a Figura 15.

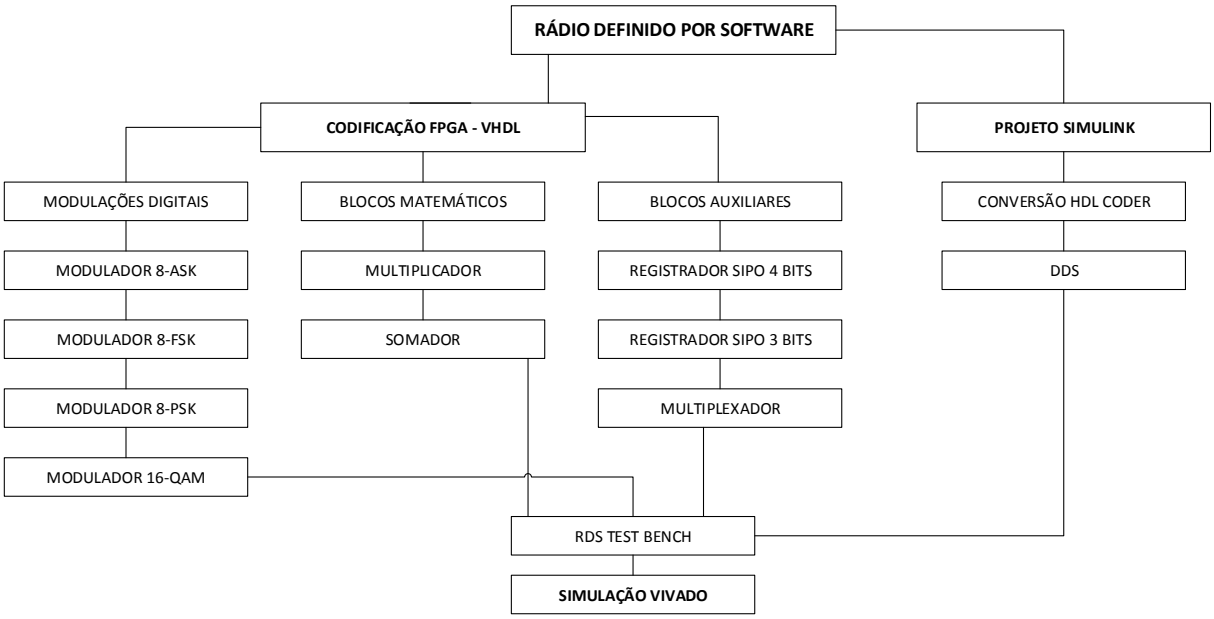

Figura 15. Fluxograma de tarefas necessárias para implementação do RDS.

3.3 *MATLAB – HDL Coder*

HDL Coder é uma ferramenta do MATLAB capaz de gerar códigos *Verilog* e VHDL, a partir de funções do MATLAB, modelos/esquemáticos do *Simulink* e gráficos do *Stateflow*. O código HDL gerado pela ferramenta pode ser usado para programação FPGA ou prototipagem ASIC [13].

Com isso, se fornece um orientador de fluxo de trabalho que automatiza a programação de circuitos FPGAS *Xilinx e Altera*. A arquitetura e a implementação do HDL é controlada pelo usuário, permitindo-o destacar os caminhos críticos e gerar estimativas de recursos de hardware. O codificador HDL fornece rastreabilidade entre o modelo *Simukink* e o código *Verilog*/HDL gerado, possibilitando a verificação de código para aplicações de alta confiabilidade [13].

3.4 Kit Digilent Nexys 3

Alinhado a comparação entre os dispositivos de programação e as opções de escolha no mercado, optou-se por utilizar um kit FPGA para implementação do Rádio Definido por Software, qual atende aos requisitos de performance e capacidade do projeto, sendo ele o Kit Nexys 3, ilustrado abaixo.

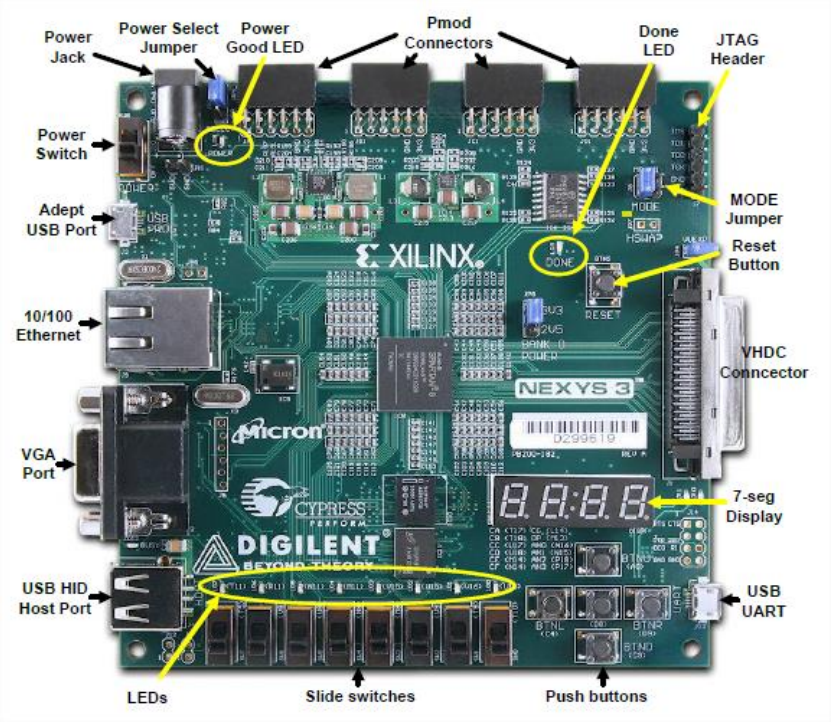

Figura 16. Kit FPGA Nexys 3 [15].

O Kit Nexys 3, é uma plataforma de desenvolvimento de circuitos digitais completa, baseada no FPGA Xilinx Spartan 6, que é otimizado para lógica de alto desempenho e oferece capacidade superior a 50% quando comparada ao seu modelo anterior. Possuindo as seguintes características [15]:

- FPGA Xilinx Spartan-6 LX26;
- RAM de 16Mbytes (x16);
- SPI 16Mbytes;
- Memória não volátil PCM pararela de 16Mbytes;
- 10/100 Ethernet;
- Porta USB 2.0 para programação e dados Xfer;
- USB-UART e USB-HID;
- Porta VGA de 8 bits;
- Oscilador CMOS de 100MHz;
- 72 I/Os para conectrores de expansão;
- GPIO incluindo 8 LEDs, 5 botões, 8 chaves e 4 displays de sete segmentos;
- Inclui cabo USB de programação;

Além dos módulos descritos acima, o *kit Nexys 3*, possui um processador RISC otimizado de 32 bits, chamado MicroBlaze, altamente configurável de fácil utilização e integração com o FPGA. Podendo ser programado em *Assembly* ou C/C++, para atender aos requisitos determinados pelo projetista [18].

3.*5 ISE Design Suite*

O software *ISE Design Suite* é o ambiente de desenvolvimento da *Xilinx*, permitindo transferir o código VHDL do projeto para a programação do dispositivo FPGA, fornecendo um ambiente adaptável a requisitos específicos [22].

Uma vez elaborado o código, utiliza-se o compilador da ferramenta para analisar o projeto em busca de erros de sintaxe e para verificação da compatibilidade entre o módulos, criando assim, informações necessárias para simulação o projeto [21].

Junto ao *ISE* é instalado o *ISim*, um simulador de linguagem de descrição de hardware (HDL), que permite executar simulações comportamentais e de temporização para VHDL, *Verilog* e projetos mistos [23].

3.6 VIVADO Design Suite

O *Vivado Design Suite* substitui o existente *ISE Design Suite* descrito anteriormente, englobando todas e incluindo novas funcionalidades em um único ambiente, quais os recursos são criados diretamente no *VIVADO,* facilitando modelos de dados escaláveis. Assim, todo o processo de projeto pode ser executado na memória sem ter que escrever ou traduzir qualquer formato de arquivo intermediário, acelerando o tempo de execução, depuração e implementação. Além disso, insere em sua platafoma de simulação, ferramentas quais permitem visualizar os dados de forma analógica, retirando a necessidade de adquirir um módulo D/A e permitindo observar os sinais modulantes e modulados em sua forma natural, justificando assim a escolha do *VIVADO* como aplicação de simulação [24].

Capítulo 4

Implementação

Nesta seção, será detalhado o processo de implementação do sistema RDS e como ocorre a construção de cada um dos seus módulos/processos, que quando integrados constituem o seguinte diagrama de bloco ilustrado pela Figura 17.

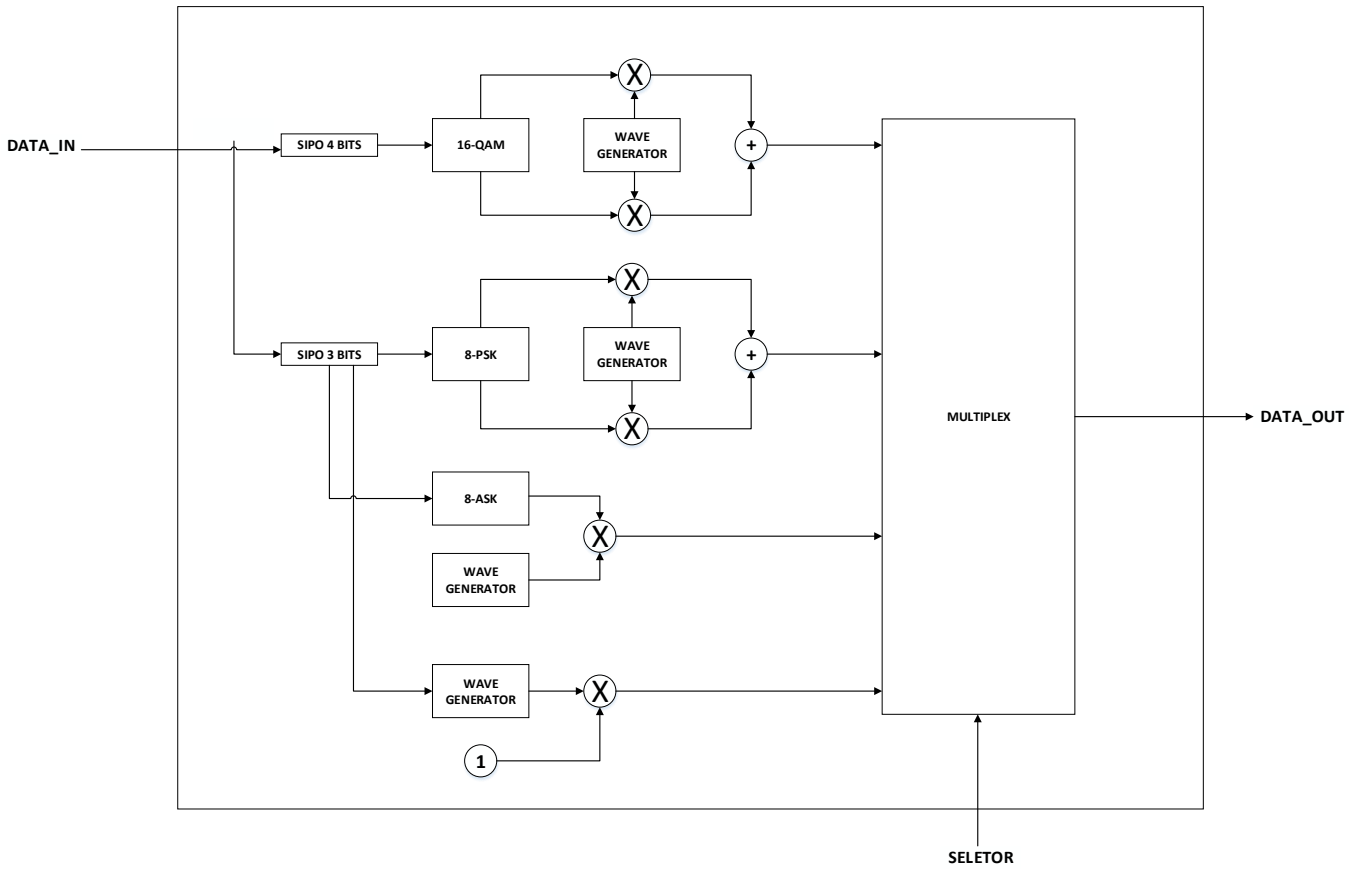

Figura 17. Diagrama de blocos RDS, visão dos módulos constituintes.

Dado o diagrama de blocos ilustrado acima, divide-se a caracterização dos blocos constituintes de acordo com as suas características comportamentais, sendo:

- DDS;
- Blocos Auxiliares (registradores e multiplexador);
- Blocos Matemáticos (somador e multiplicador);
- Moduladores Digitais;

4.1 DDS

4.1.1 Projeto *Simulink – HDL Coder*

Os algoritmos e projetos usados para definir sistemas são geralmente modelados usando linguagens de alto nível *como MATLAB, MATLAB - HDL Coder*, C ou C++, porém tais projetos podem não ser adequados para implementação em hardware real. O *MATLAB HLD-Coder* como descrito anteriormente é uma ferramenta que pode ser usada para converter projetos modelados no *Simulink* ou *Stateflow* para códigos VHDL, passíveis de serem implementados em ASIC ou FPGA. Usando tal ferramenta, arquitetos de sistemas e projetistas podem gastar mais tempo aprimorando modelos e algoritmos através de uma prototipagem rápida, gastando menos tempo na codificação HDL. O codificador gera o código VHDL ou *Verilog* que implementa o projeto embarcado no modelo.

Para desenvolvimento do projeto proposto, utilizou o MATLAB HDL – CODER para construir um diagrama de blocos capaz de gerar sinais senoidais em quadratura, nesse caso, onda Seno e onda Cosseno, assim como indicado na figura abaixo.

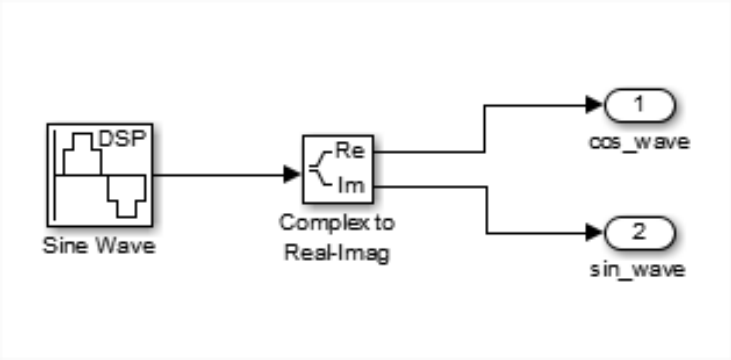

Figura 18. Esquemático Simulink HDL – Coder, Gerador de ondas.

Analisando os blocos contituintes do esquemático acima, utilizou-se os blocos *Sine Wave e Complex to Real-Imag*. O bloco *Sine Wave* é responsável por gerar, ler e mostrar esquemas de senoide, respeitando os parâmetros de configuração indicados pela figura a seguir.

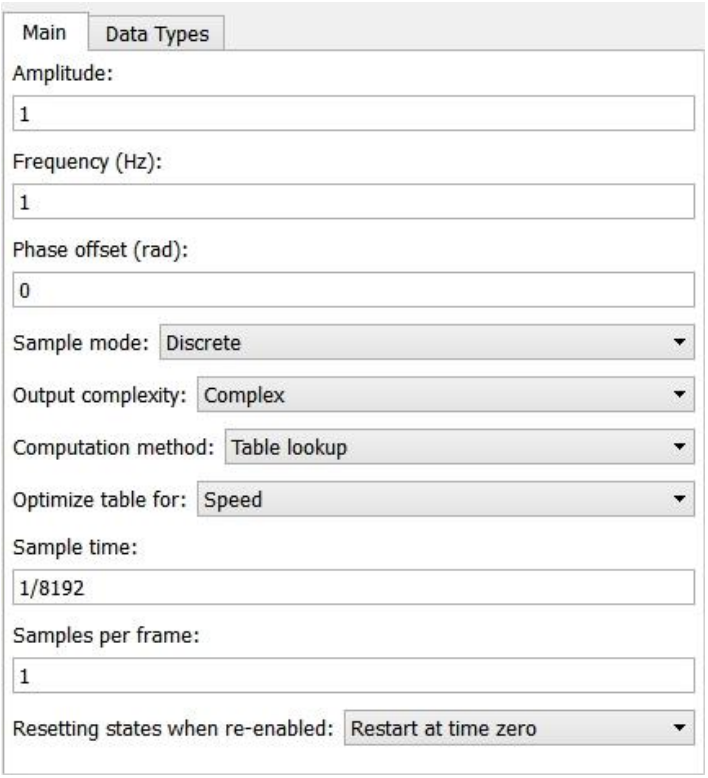

Figura 19. Janela de configuração do bloco *Sine Wave*.

Analisando a janela de configuração ilustrada, vale destacar a definição dos campos *Sample time, Output complexity e Computation method. Sample time*, faz referência a uma relação que determina a quantidade de amostras utilizadas para representar a onda de saída, sendo que assim como configurado, tem-se um seno/cosseno com 4096 amostras. Já o campo *Output complexity*, é responsável por determinar o tipo do dado obtido na saída, de modo que, como configurado se tem a saída com valores complexos, o que leva a utilização do bloco subsequente. Por fim, o campo *Computation method* faz alusão ao método de computar/gerar a onda, onde como configurado faz uso da técnica *Table lookup,* ilustrando o mesmo método utilizado pelo DDS, o que faz com que o mesmo atue como o sintetizador de forma de onda.

Assim como descrito anteriormente, o campo *Output complexity* definido em *Complex*, gera a necessidade do uso do bloco subsequente *Complex to Real-Imag*, qual é responsável por converter o sinal complexo de entrada em dois sinais distintos na saída, valores reais e imaginários. Realizar tal conversão, é em suma gerar ondas senoidais em quadratura, de modo que, a onda representada pelos

valores reais reproduz o Cosseno e a onda representada pelos valores imaginários reproduz o Seno.

Dado a construção, simulação e validação do esquemático acima, aplica-se a funcionalidade da ferramenta *HDL-Coder*, convertendo o esquemático para código VHDL, o qual é aperfeiçoado na ferramenta ISE Project *Navigator* da *Xilinx*.

4.1.2 Projeto Codificação VHDL - ISE

Dada a conversão do esquemático do *Simulink* para código VHDL, se obtém um código inicial, denominado *Wave Generator*, para prosseguir com a proposta do Sintetizador. De forma com que, pode-se verificar que tal código gerado pela plataforma do MATLAB, não atendia aos requisitos do projeto, o qual requer com que as formas de onda de saída variem em amplitude e frequência.

Assim, adicionou-se ao mesmo uma rotina responsável pela variação da frequência, controlada por uma variável de entrada definida pelo registrador SIPO, definindo o Modulador 8-FSK, ambos descritos posteriormente.

Ainda dentro do código *Wave Generator*, vale destacar o comportamento do processo *Wave Lookup*, responsável por atribuir um valor de saída para cada endereço gerado pelo processo *Address Counter*, dada a frequência determinada pela variável de entrada. O código em questão, *Wave Generator*, é descrito no APÊNCICE B, esse, porém com um número menor de amostras (10) para um melhor entendimento e visualização do mesmo. Por fim, destaca-se que os sinais de saída do módulo são representados por vetores de 12 bits.

4.2 Blocos auxiliares

4.2.1 Registrador SIPO 4 Bits

Codificação VHDL responsável por agrupar os dados de entrada do RDS em palavras de 4 bits, que posteriormente são transferidas para entrada do Modulador 16-QAM, afim de determinar o símbolo a ser transmitido pelo canal. É construído em cima do conceito de registradores de deslocamento SIPO, através de uma rotina de processamento que recebe o dado da entrada e desloca-o para a posição menos significativa de um vetor a cada ciclo de *clock*, fazendo isso recursivamente com os

dados de entrada até preencher as 4 posições do vetor, que por fim é encaminhado para a saída do bloco de maneira contínua, ou seja, muda-se entrada, altera-se a saída. Observa-se as características de projeto desse bloco mediante o código descrito no APÊNDICE G.

4.2.2 Registradores SIPO 3 Bits

Codificação VHDL responsável por agrupar os dados de entrada do RDS em palavras de 3 bits, que posteriormente são transferidas para entrada dos Moduladores 8-ASK, 8-FSK e 8-PSK, afim de determinar o símbolo a ser transmitido pelo canal. Também é construído em cima do conceito de registradores de deslocamento SIPO, que tem sua caracterização assim como descrito anteriormente. Observa-se as características de projeto desse bloco mediante o código descrito no APÊNDICE H.

4.2.3 Multiplexador

Bloco responsável por receber os sinais referentes a cada tipo de modulação entregue pelo RDS, e amostrar em seu canal de saída somente um deles, baseado na variável de seleção determinada pelo usuário. É construído em cima de uma rotina de processamento que cria relações entre a seleção do usuário e os sinais modulados, assim como indicado na Tabela 4.

Seleção	Saída
00	Sinal modulado em 16-QAM
01	Sinal modulado em 8-PSK
10	Sinal modulado em 8-ASK
11	Sinal modulado em 8-FSK

Tabela 4. Relação de amostragem do Multiplexador.

A partir das informações da Tabela 4, codifica-se o módulo VHDL de modo a criar essa relação, onde a seleção é representada por uma variável de entrada de 2 bits, os sinais modulados são representados por variáveis de entrada de 17 bits e

por fim uma variável de saída de 17 bits, que atribui em si o sinal determinado na seleção, assim como ilustrado no APÊNDICE I.

4.3 Blocos matemáticos

Projetados em cima das funções entregues pelas bibliotecas IEEE.NUMERIC_STD.ALL e IEEE.STD_LOGIC_SIGNED.ALL, apresentam baixa complexidade de projeto alinhado ao objetivo de auxiliar na construção das modulações entregues pelo RDS.

4.3.1 Multiplicador

Codificação VHDL responsável por multiplicar dois sinais de entrada e apresentar o resultado no sinal de saída.

O bloco possui duas entradas, “wave” (12 bits), que faz referência ao sinal gerado pelo bloco *Wave Generator* e “amplitude” (5 bits) que faz alusão ao sinal de amplitude fornecido por uma das amplitudes entregues pelo RDS, que quando multiplicadas deslocam para sua saída, “result” (17 bits), o resultado da operação, assim como descrito no APÊNDICE E, tal bloco lógico faz referência direta as modulações ASK e FSK e indireta as modulações PSK e QAM.

4.3.2 Somador

Codificação VHDL responsável por somar dois sinais de entrada e apresentar o resultado no sinal de saída.

Destaca-se que o bloco matemático somador é utilizado apenas na construção das modulações 8-PSK e 16-QAM, que apresentam componentes em seno e cosseno que precisam ser somadas afim de se obter o sinal modulado. Assim no APÊNDICE F, podemos observar que o bloco possui duas entradas, “X” e “Y” (17 bits), que fazem referência aos sinais componentes em seno e cosseno, respectivamente, e a saída “S” (17 bits) que faz alusão ao sinal final modulado. Por fim, destaca-se que o bloco faz tanto a soma de sinais positivos quanto negativos através do uso da biblioteca IEEE.STD_LOGIC_SIGNED.ALL.

4.4 Moduladores Digitais

4.4.1 Modulador 8-ASK

Dada a modulação 8-ASK, sabe-se que essa modula símbolos de 3 bits, utilizando assim 8 amplitudes diferentes na construção do seno, sendo que cada amplitude é equivalente a transmissão de um símbolo, relação esta representada na Tabela 5.

	Amplitude	
Símbolos	Decimal	Binário
S0 – 000	0	00000
S1 – 001	2	00010
S2 – 010	4	00100
S3 – 011	6	00110
S4 – 100	8	01000
S5 – 101	10	01010
S6 – 110	12	01100
S7 – 111	14	01110

Tabela 5. Relação entre símbolos e amplitudes da modulação 8-ASK.

Tendo as informações da Tabela 5, codifica-se o módulo VHDL de modo a fazer esse casamento de símbolo com amplitude, através de um processo de verificação que determina qual símbolo deve ser transmitido, onde o símbolo é representado por uma variável de entrada de 3 bits e a amplitude do seno representada por um dado de saída de 5 bits, assim como ilustrado no APÊNDICE A.

4.4.2 Modulador 8-FSK

Compreende-se que a modulação 8-FSK, modula símbolos de 3 bits, utilizando assim 8 frequências diferentes na construção do seno, sendo que cada frequência é equivalente a transmissão de um símbolo. Sabendo que a frequência

natural é delimitada pelo *clock* inserido no sistema, cria-se uma relação entre os símbolos e frequências derivadas da frequência natural, representada essa na Tabela 6.

Símbolos	Fator multiplicativo da frequência natural
S0 - 000	x1
S1 - 001	x2
S2 - 010	x3
S3 - 011	x4
S4 - 100	x5
S5 - 101	x6
S6 - 110	x7
S7 - 111	x8

Tabela 6. Relação entre símbolos e frequências da modulação 8-FSK.

Tendo as informações da Tabela 6, aproveita-se do código referente ao DDS e adiciona ao mesmo uma rotina responsável pela variação da frequência, controlada por uma variável definida pelo registrador SIPO de 3 bits, qual representa o símbolo a ser transmitido. Tal funcionalidade é aplicada junto ao processo de geração dos endereços, *Address Counter*, sabendo-se que os endereços são gerados de forma sequencial, dado um incremento “sum_freq”, e que esse valor está dentro do intervalo [1,8] definido pela rotina anterior, para cada valor dentro do intervalo o processo vai percorrer as posições de endereços em velocidades distintas, gerando 8 frequências de transmissão diferentes, qual cada uma corresponde a um sinal derivado da frequência natural dada um fator multiplicativo, ou seja, o incremento, assim como descrito no APÊNDICE B.

4.4.3 Modulador 8-PSK

Modulação 8-PSK modula símbolos de 3 bits, utilizando 8 fases diferentes para a representação de cada símbolo transmitido, assim como ilustrado na Figura 20, qual mostra o sinal como um diagrama de dispersão bidimensional no plano

complexo em instantes de amostragem de símbolo, onde em cada instante o símbolo ocupa apenas uma posição no diagrama.

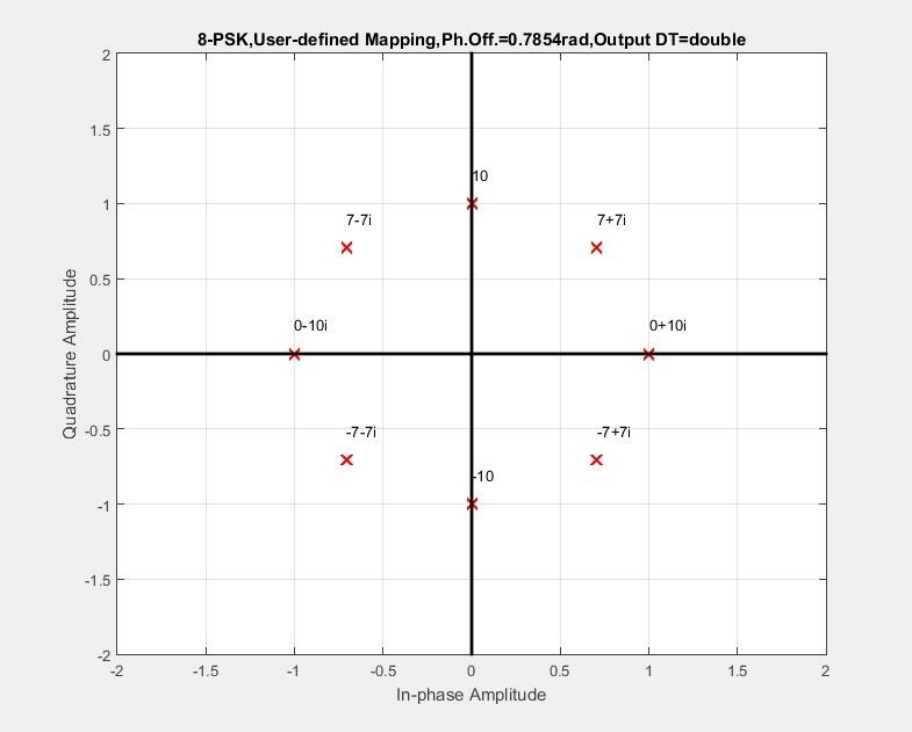

Figura 20. Constelação 8-PSK.

A partir da Figura 20, constrói-se a Tabela 7, a qual representa os valores de amplitude dos sinais seno e cosseno, baseado nos pares de coordenadas de cada símbolo.

	Amplitude	
Símbolos	Cosseno (eixo real)	Seno (eixo imaginário)
S0 – 000	0	10
S1 – 001	7	7
S2 – 010	10	0
S3 – 011	7	-7
S4 – 100	0	-10
S5 – 101	-7	-7
S6 – 110	-10	0

S7 – 111	-7	7

Tabela 7. Relação entre símbolos e amplitudes da modulação 8-PSK.

A partir das informações da Figura 20 e Tabela 7, codifica-se o módulo VHDL de modo a fazer esse casamento entre símbolo e fase, através de um processo de verificação que determina qual símbolo deve ser transmitido, atribuindo as variáveis de saída, as amplitudes de seno e cosseno referente àquela posição ocupada pelo símbolo na constelação. O símbolo é representado por uma variável de entrada de 3 bits enquanto as amplitudes do seno e do cosseno são representadas por variáveis de saída de 5 bits, assim como ilustrado no APÊNDICE C.

4.4.4 Modulador 16-QAM

Modulação 16-QAM modula símbolos de 4 bits, utilizando 16 combinações de fase e amplitude diferentes para a representação de cada símbolo transmitido, assim como ilustrado na Figura 21, qual mostra o sinal como um diagrama de dispersão bidimensional no plano complexo em instantes de amostragem de símbolo, onde em cada instante o símbolo ocupa apenas uma posição no diagrama.

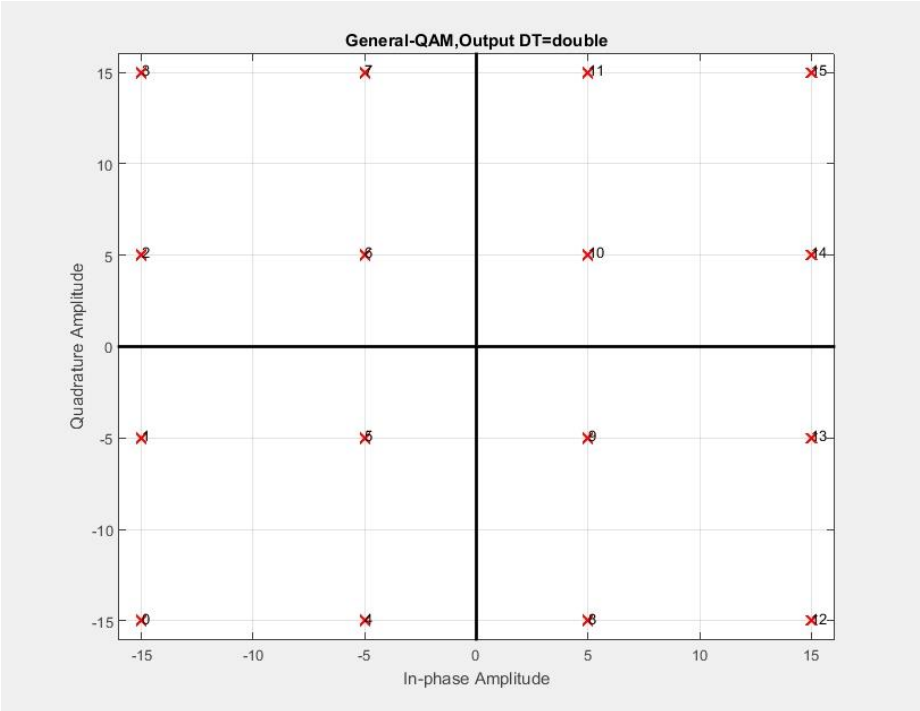

Figura 21. Constelação 16-QAM.

A partir da Figura 21, constrói-se a Tabela 8, a qual representa os valores de amplitude dos sinais seno e cosseno, baseado nos pares de coordenadas de cada símbolo.

	Amplitude	
Símbolos	Cosseno (eixo real)	Seno (eixo imaginário)
S0 – 0000	-15	15
S1 – 0001	-15	5
S2 – 0010	-15	-5
S3 – 0011	-15	-15
S4 – 0100	-5	15
S5 – 0101	-5	5
S6 – 0111	-5	-5
S7 – 0111	-5	-15
S8 – 1000	5	15
S9 – 1001	5	5
S10 – 1010	5	-5
S11 – 1011	5	-15
S12 – 1100	15	15
S13 – 1101	15	2
S14 – 1110	15	-5
S15 – 1111	15	-15

Tabela 8. Relação entre símbolos e amplitudes da modulação 16-QAM.

A partir das informações da Figura 21 e Tabela 8, codifica-se o módulo VHDL de modo a fazer esse casamento entre símbolos, fase e amplitude, através de um processo de verificação que determina qual símbolo deve ser transmitido, atribuindo as variáveis de saída, as amplitudes de seno e cosseno referente àquela posição ocupada pelo símbolo na constelação. O símbolo é representado por uma variável de entrada de 4 bits enquanto as amplitudes do seno e do cosseno são

representadas por variáveis de saída de 5 bits, assim como ilustrado no APÊNDICE D.

Capítulo 5

Resultados e Discussões

.

Para realizar a integração dos módulos codificados, faz-se necessário a construção de um novo módulo, denominado RDS, onde nesse é feita toda a definição estrutural do projeto, constando todas as entradas, saídas e sinais de ligação do circuito. O módulo realiza essa integração através de instanciações, declarações e atribuições, gerando assim as estruturas operacionais do mesmo, assim como observado no APÊNDICE J.

Assim, pode-se definir que o RDS implementado é construído através da integração dos seguintes módulos/processos:

- Registrador SIPO de 4 bits;
- Registrador SIPO de 3 bits;
- Modulador 8-ASK;
- Modulador 8-FSK;
- Modulador 8-PSK;
- Modulador 16-QAM;
- *Wave Generator*;
- Multiplicador;
- Somador;
- Multiplexador;

Para melhor visualização do circuito projetado, segue figura referente ao esquemático RTL gerado pela ferramenta ISE, representando o sistema RDS.

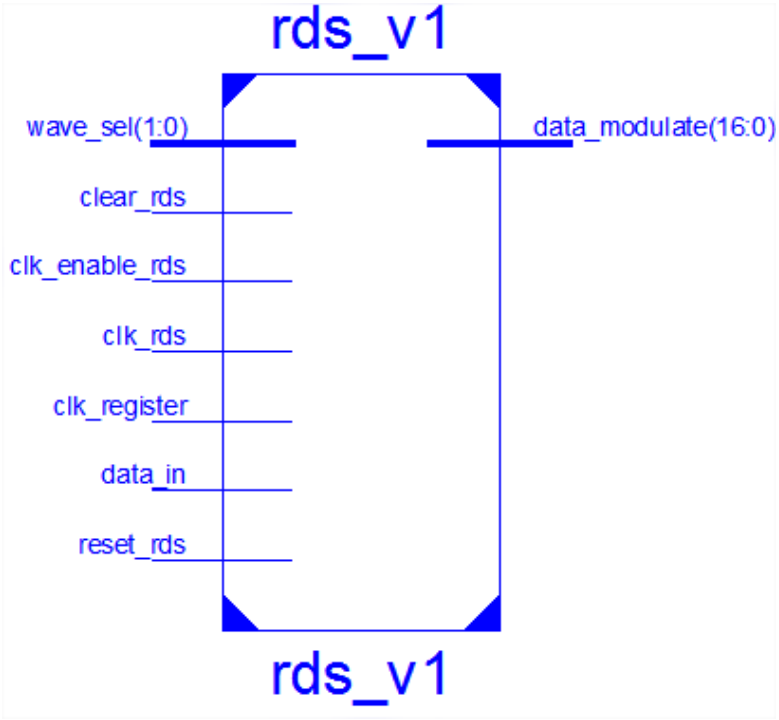

Figura 22. Esquemático RTL RDS.

Analisando as Figuras 22 e 17, a última ilustrada no capítulo anterior, tem-se uma visão da arquitetura utilizada no projeto, a primeira um bloco lógico fechado, trazendo as variáveis de entrada e saída do RDS, e a segunda um diagrama de blocos da solução, onde verifica-se que as modulações 8-PSK e 16-QAM foram construídas através da soma das suas componentes em seno e cosseno, enquanto 8-ASK e 8-FSK, são constituídas em blocos individuais usando apenas uma componente do bloco *Wave Generator*. Caracteriza-se as seguintes variáveis do bloco RDS:

- data_in – entrada – 1 bit: responsável por receber o dado digital de entrada a ser transmitido.
- clk_rds – entrada- 1 bit: clock de funcionamento do RDS.
- clk_register – entrada – 1 bit: clock de funcionamento dos registradores.
- clk_enable_rds – entrada – 1 bit: responsável por habilitar o processo de geração de endereços do bloco *Wave Generator*.
- reset_rds – entrada – 1 bit: responsável por resetar o processo de geração de endereços do bloco Wave Generator.
- clear_rds – entrada – 1bit: responsável por limpar, ou seja, atribuir 0 em todas as posições da saída dos blocos registradores.

- wave_sel – entrada – 2 bits: responsável por selecionar a modulação a ser utilizada no RDS.
- data_modulate – saída – 17 bits: responsável por atribuir em si o sinal da modulação setada pela variável anterior.

Após síntese do código VHDL do RDS, incluindo todos os módulos, em cima da plataforma de desenvolvimento ISE, essa retorna informações referentes a ocupação dos recursos disponíveis na placa, assim como ilustrada na Figura 23.

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	39	18224	0%
Number of Slice LUTs	2686	9112	29%
Number of fully used LUT-FF pairs	29	2696	1%
Number of bonded IOBs	25	232	10%
Number of BUFG/BUFGCTRLs	2	16	12%
Number of DSP48A1s	5	32	15%

Figura 23. Tabela indicativa de consumo/ocupação de recursos da placa *Digilent Nexys 3.*

Analisando os dados referentes a Figura 23, pode-se aferir que não teve consumo excessivo de nenhum dos recursos da placa, mantendo todos abaixo dos 30%.

5.1 Simulações

Dada a ferramenta de codificação VHDL, ISE, e usando a plataforma de simulação comportamental do VIVADO, é possível analisar o comportamento do circuito RDS projetado. Fazendo necessário a configuração de um script test bench, responsável por emular e determinar as variáveis de entrada do circuito, sendo esse descrito pelo APÊNCIDE K.

Em paralelo, afim de avaliar e discutir os resultados obtidos na simulação do VIVADO, utilizou-se do ambiente de simulação do MATLAB, *Simulink,* para criar, testar e comparar moduladores com as mesmas características comportamentais dos blocos projetados em VHDL.

Para análise do modulador 8-ASK, desenvolveu-se o seguinte diagrama ilustrado pela Figura 24.

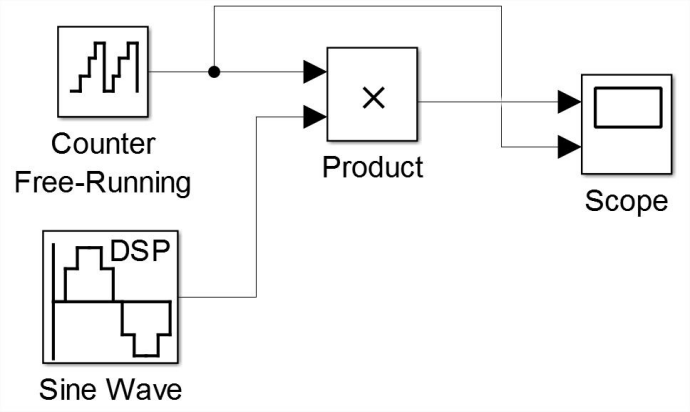

Figura 24. Diagrama de blocos Modulação 8-ASK, ambiente *Simulink.*

Já para análise do modulador 8-FSK, desenvolveu-se o seguinte diagrama ilustrado pela Figura 25.

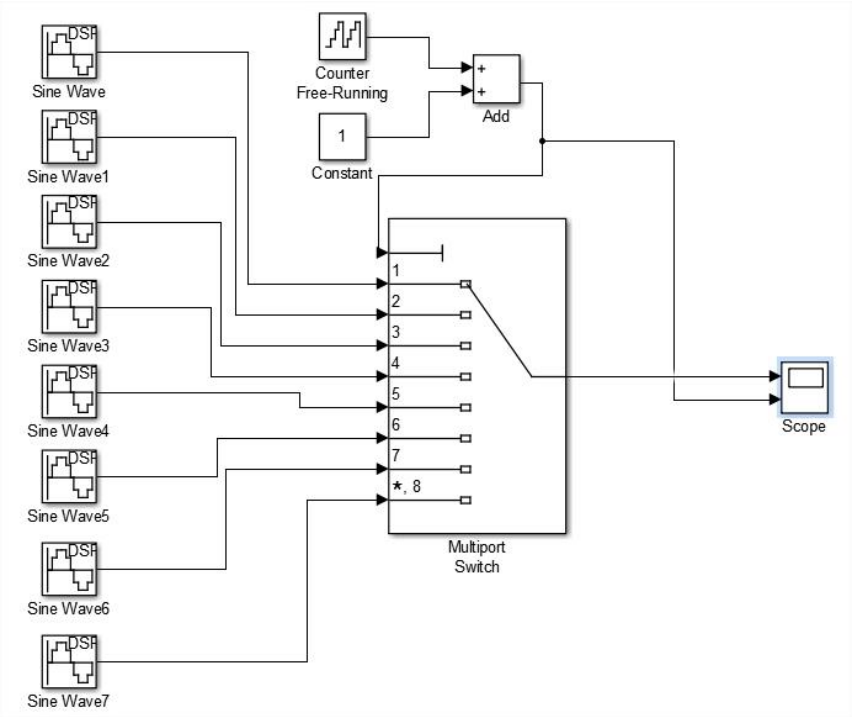

Figura 25. Diagrama de blocos Modulação 8-FSK, ambiente *Simulink.*

Para verificação do modulador 8-PSK, desenvolveu-se o seguinte diagrama ilustrado pela Figura 26.

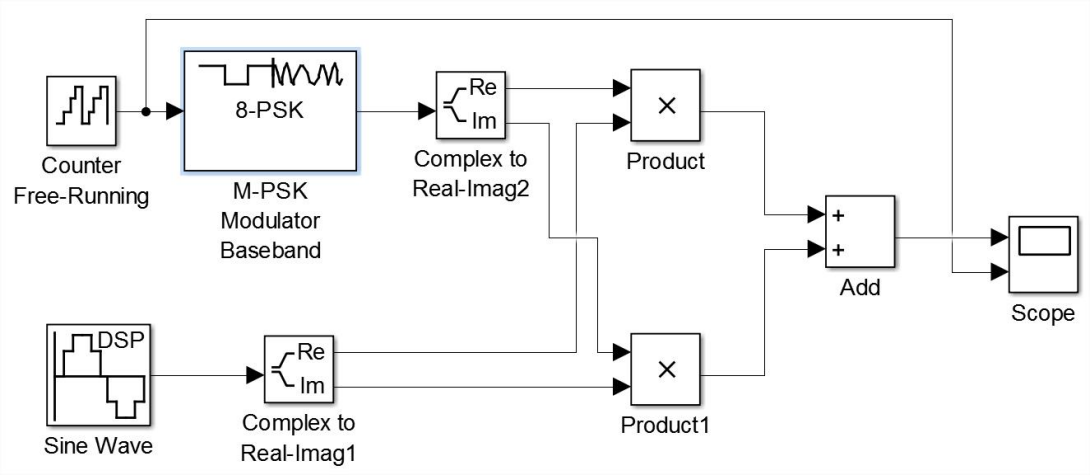

Figura 26. Diagrama de blocos Modulação 8-PSK, ambiente *Simulink*.

Por fim, para verificação do modulador 16-QAM, desenvolveu-se o seguinte diagrama ilustrado pela Figura 27.

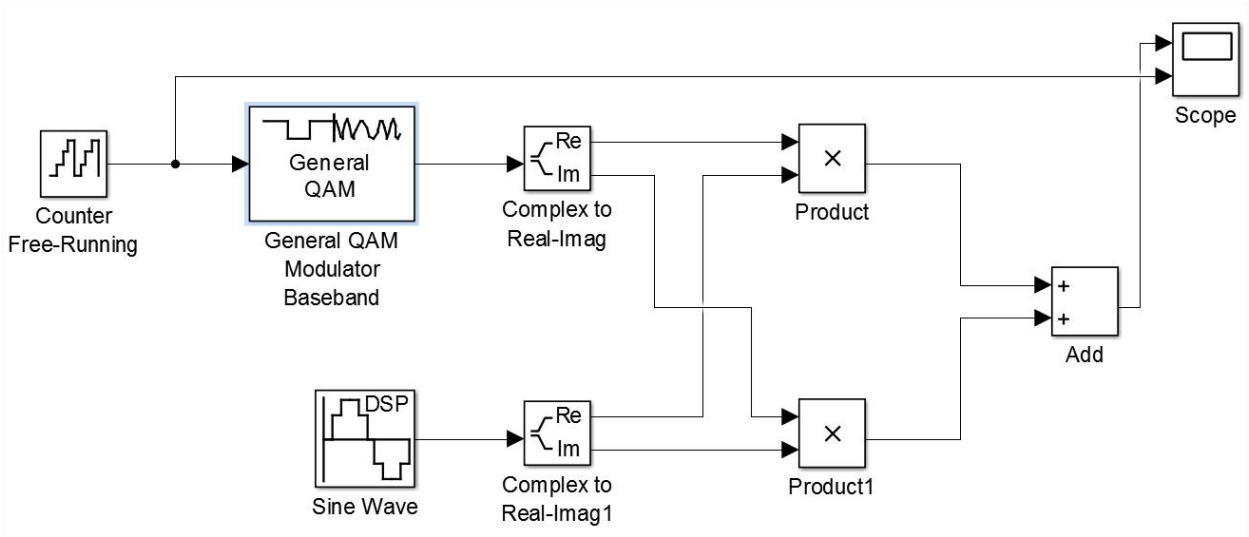

Figura 27. Diagrama de blocos Modulação 16-QAM, ambiente *Simulink*.

Para a visualização comportamental do RDS, fez-se a seguintes simulações indicadas na tabela, tanto no ambiente VIVADO quanto MATLAB-Simulink.

Cenário	Saída
1	Sinal modulado em 8-ASK
2	Sinal modulado em 8-FSK
3	Sinal modulado em 8-PSK
4	Sinal modulado em 16-QAM

Tabela 9. Parâmetros/rotinas de Simulação do RDS.

Dada as especificações do cenário de simulação 1, obtém-se as Figura 28 e 29.

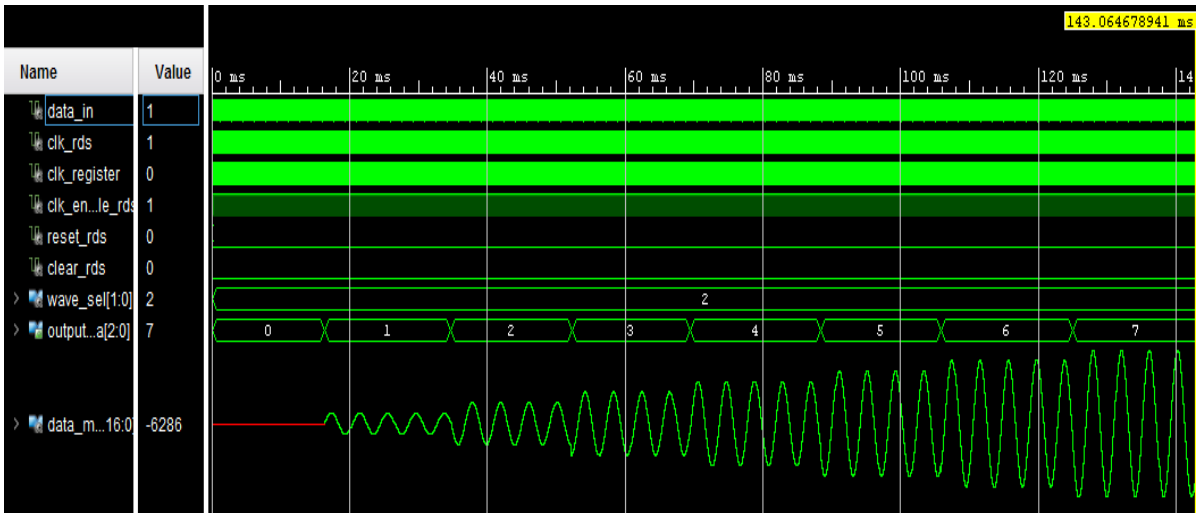

Figura 28. Simulação da Modulação 8-ASK, ilustrando todas as transições de amplitude em cima do sinal modulado.

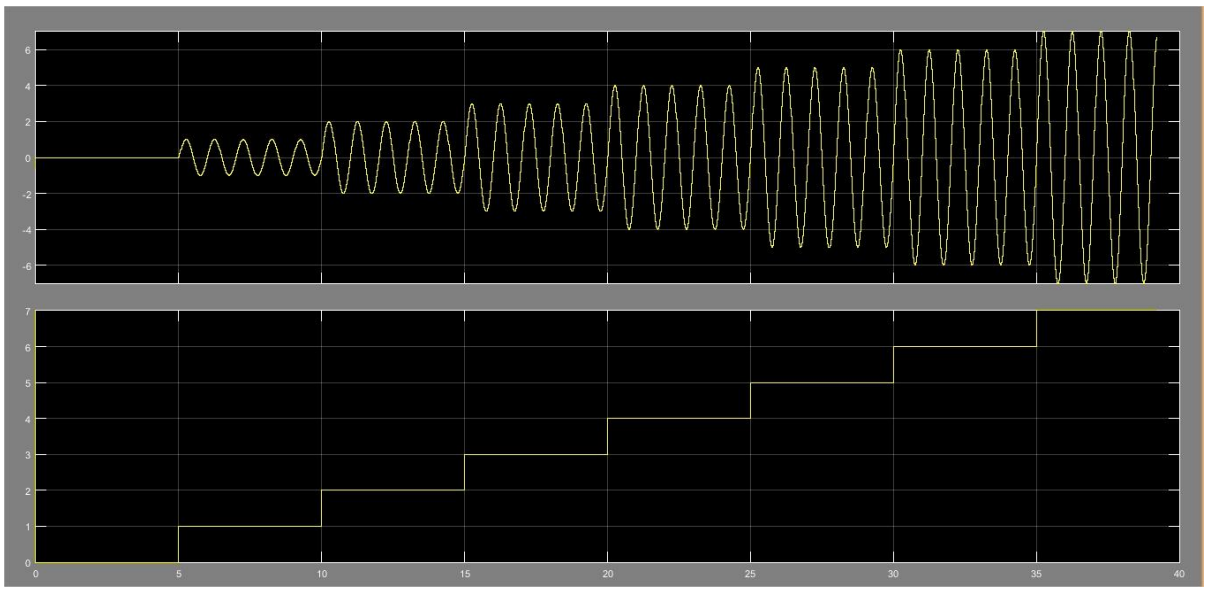

Figura 29. Simulação da Modulação 8-ASK, ilustrando todas as transições de amplitude em cima do sinal modulado – ambiente *Simulink*.

Dada a Figura 28, pode-se observar que para cada valor agrupado pelo registrador SIPO de 3 bits, o modulador transmite essa palavra em cima de um sinal senoidal com uma dada amplitude característica, ou seja, para cada palavra tem-se

um sinal de amplitude distinta, assim como esperado. Porém nota-se um problema de temporização entre as transições, onde o modulador é incapaz de completar o ciclo do seno atual para que inicie a transmissão de uma outra palavra, em uma outra amplitude, sofrendo uma transição abrupta. Comparando com a Figura 29, pode-se aferir que o comportamento da onda do sinal modulado em ambos os ambientes de desenvolvimento, exibe resultados semelhantes, validando assim o projeto do Modulador 8-ASK codificado em VHDL. Ressaltando que a única diferença entre as figuras é referente ao erro de temporização, que não é observado no ambiente MATLAB-*Simulink.*

Dada as especificações do cenário de simulação 2, obtém-se as Figura 30 e 31.

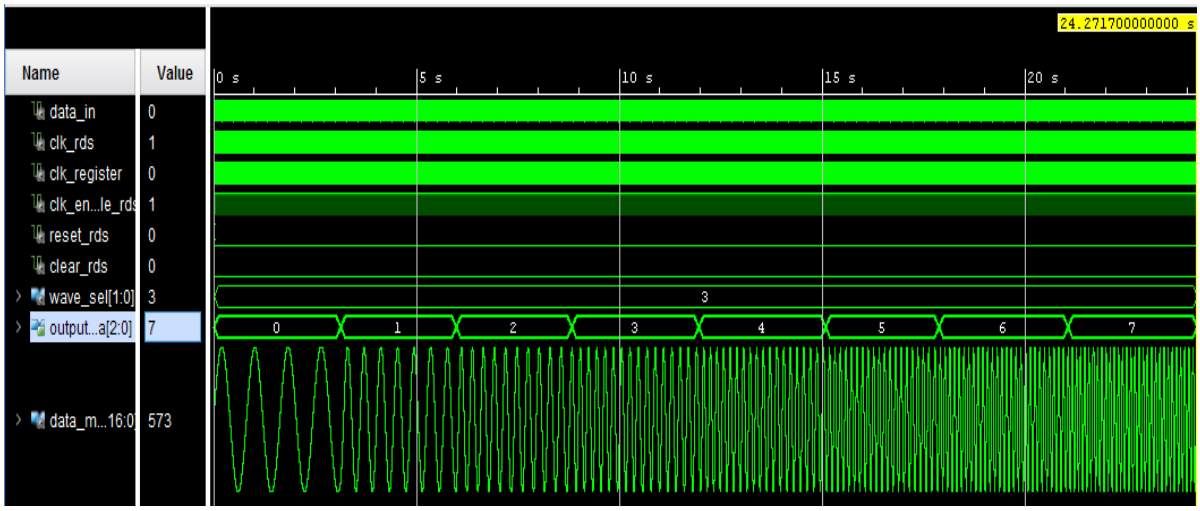

Figura 30. Simulação da Modulação 8-FSK, ilustrando todas as transições de frequência em cima do sinal modulado.

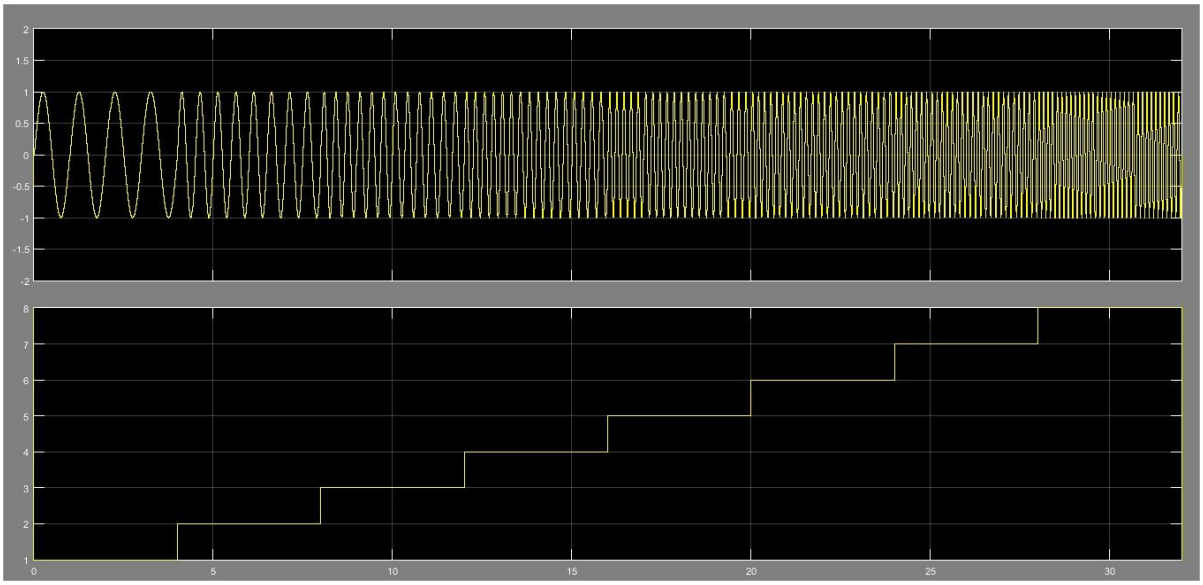

Figura 31. Simulação da Modulação 8-FSK, ilustrando todas as transições de frequência em cima do sinal modulado – ambiente *Simulink*.

Analisando a Figura 30 pode-se observar que o modulador se comportou assim como esperado, tendo para cada palavra um sinal de frequência distinta. Porém nota-se o mesmo problema de temporização descrito anteriormente. Comparando com a Figura 31, pode-se aferir que o comportamento da onda do sinal modulado em ambos os ambientes de desenvolvimento, exibe resultados semelhantes, validando assim o projeto do Modulador 8-FSK codificado em VHDL. Lembrando apenas que como no caso anterior, o erro de temporização não é observado no ambiente *MATLAB-Simulink*.

Dada as especificações do cenário de simulação 3, obtém-se as Figura 32 e 33.

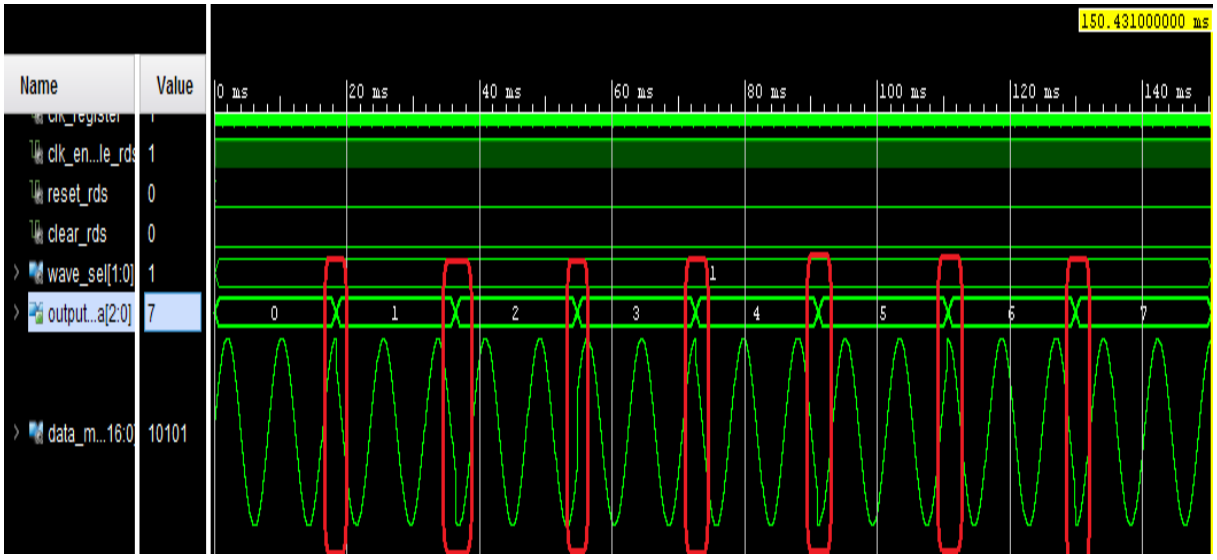

Figura 32. Simulação da Modulação 8-PSK, ilustrando todas as transições de fase em cima do sinal modulado.

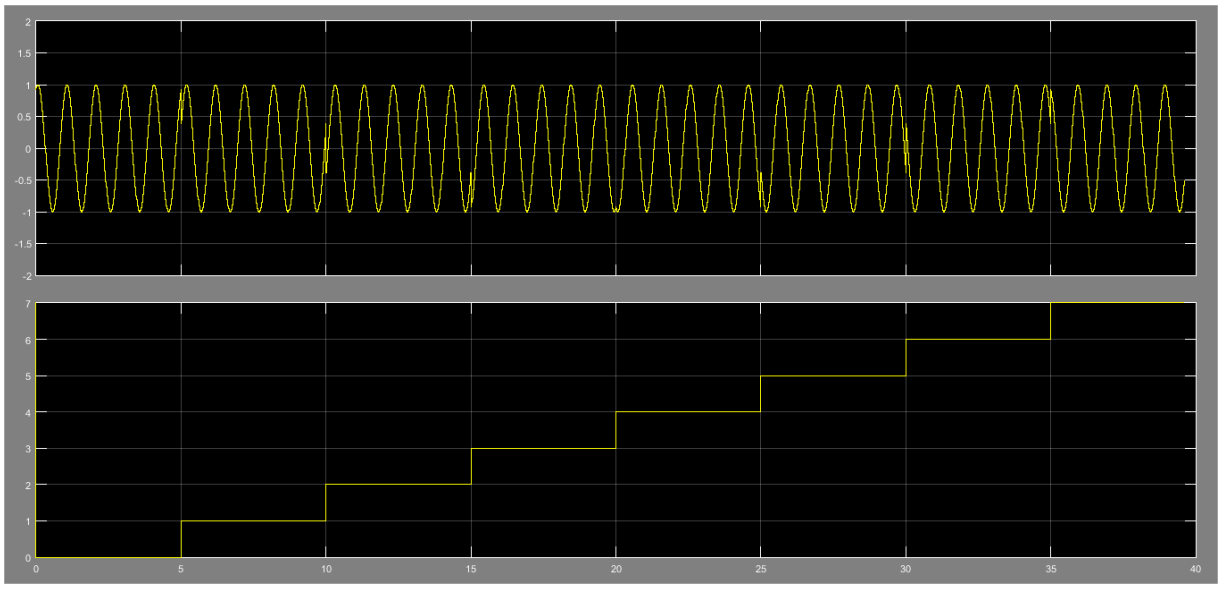

Figura 33. Simulação da Modulação 8-PSK, ilustrando todas as transições de fase em cima do sinal modulado – ambiente *Simulink*.

Considerando a Figura 32 pode-se observar que o modulador se comportou assim como esperado, tendo para cada palavra um sinal de fase distinta. Porém nota-se o mesmo problema de temporização descrito no primeiro cenário. Comparando com a Figura 33, pode-se aferir que o comportamento da onda do sinal modulado em ambos os ambientes de desenvolvimento, exibe resultados semelhantes, validando assim o projeto do Modulador 8-PSK codificado em VHDL.

Destaca-se que como verificado no primeiro cenário, o erro de temporização não é observado no ambiente MATLAB-*Simulink*.

Dada as especificações do cenário de simulação 4, obtém-se as Figuras 34 e 35.

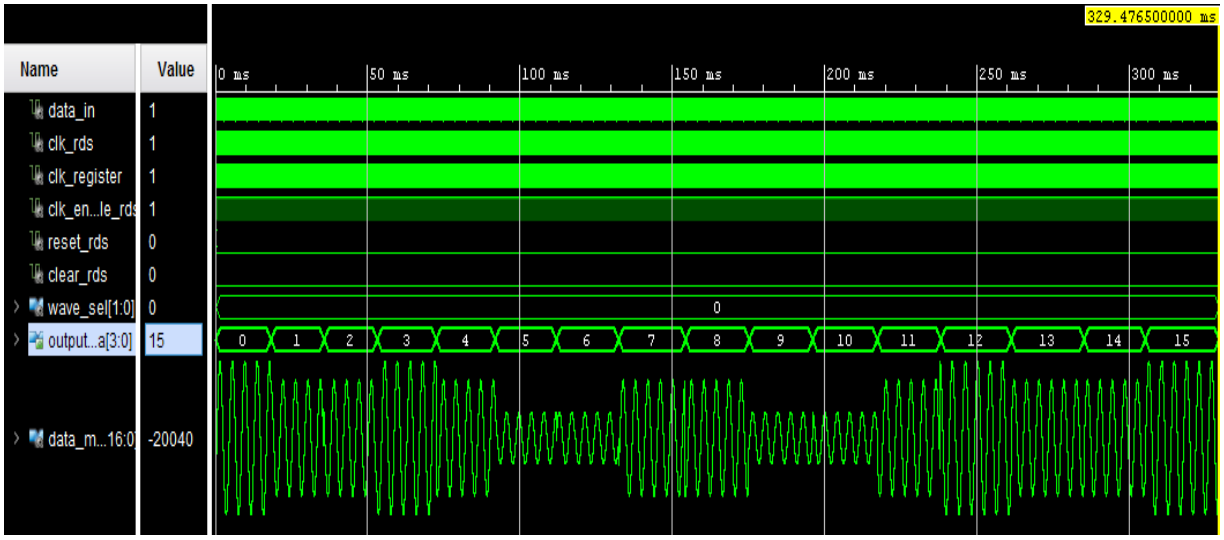

Figura 34. Simulação da Modulação 16-QAM, ilustrando todas as transições de fase e amplitude em cima do sinal modulado.

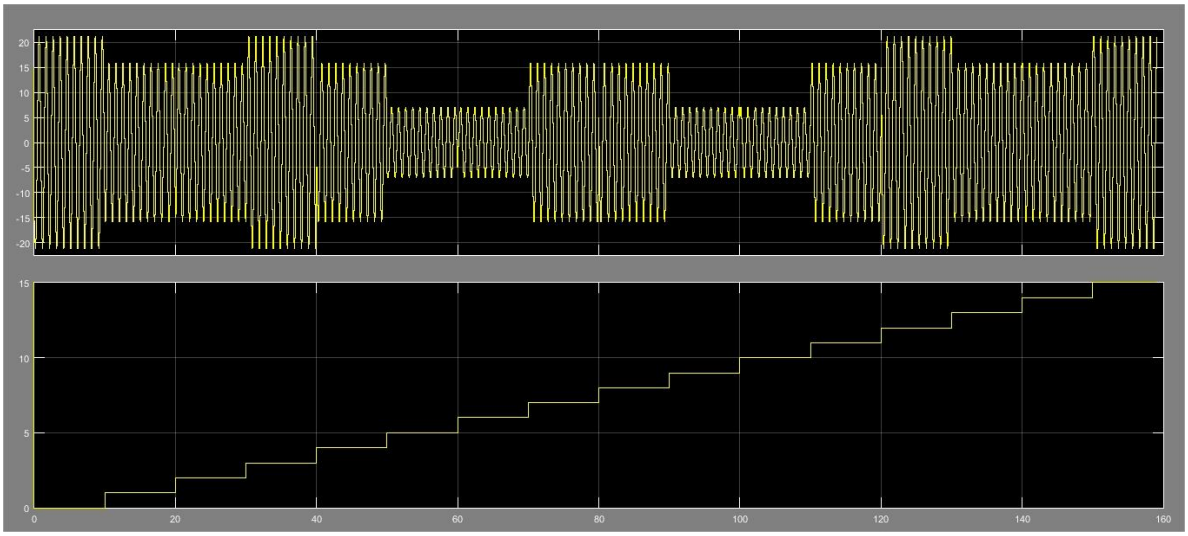

Figura 35. Simulação da Modulação 16-QAM, ilustrando todas as transições de fase e amplitude em cima do sinal modulado – ambiente *Simulink*.

Por fim, explorando a Figura 34, pode-se observar que para cada valor agrupado pelo registrador SIPO de 4 bits, o modulador se comportou assim como

esperado, tendo para cada palavra um sinal de fase e amplitude distinta. Porém nota-se o mesmo problema de temporização descrito no primeiro cenário. Comparando com a Figura 35, pode-se aferir que o comportamento da onda do sinal modulado em ambos os ambientes de desenvolvimento, exibe resultados semelhantes, validando assim o projeto do Modulador 16-QAM codificado em VHDL. Lembrando apenas que como verificado no primeiro cenário, o erro de temporização não é observado no ambiente *MATLAB-Simulink*.

5.2 Análises de Operação

A plataforma de desenvolvimento ISE, através de relatórios de síntese e implementação, possibilita a visualização de dados referentes ao *timing* de funcionamento do circuito, sendo eles:

- *Minimum period:* 5.068ns
- *Maximum Frequency:* 197.315 MHz
- *Minimum input arrival time before clock:* 3.356ns
- *Maximum output required time after clock:* 22.843ns
- *Maximum combinational path delay:* 6.839ns

Analisando os dados listados acima, pode-se aferir que o RDS implementado em questão atende ao requisito da faixa de operação de frequências e que os valores de *delays* inerentes às conexões do circuito estão dentro do esperado, fato que não impacta no processo de transmissão dos dados.

Capítulo 6

Conclusões

6.1 Considerações Finais

Para o projeto, foi implementado um DDS usando a integração das ferramentas de projeto *MATLAB HDL-Coder, ISE and VIVADO Xilinx*, validando assim um ambiente integrado de desenvolvimento e simulação, qual otimiza o tempo gasto em cima da codificação VHDL, dando mais prazo para o aprimoramento de modelos e algoritmos.

Dado o circuito DDS, prosseguiu-se com a codificação VHDL das modulações digitais, dos blocos matemáticos e blocos auxiliares, que quando agrupados, constituem o RDS, qual pode-se aferir resultados satisfatórios das simulações de implementação em hardware, em cima da plataforma do *VIVADO*, dado que as modulações estão funcionando tão corretamente quanto o seu modelo paralelo desenvolvido no MATLAB. Ressaltando que a única diferença entre as simulações é referente ao erro de temporização, que não é observado no ambiente *MATLAB-Simulink*, mas que pode ser solucionado através de rotinas de processamento que quando incorporadas aos moduladores sejam capazes de identificar quando se inicia e quando se encerra a transmissão de um símbolo.

O RDS é uma ferramenta de alta aplicabilidade no campo das telecomunicações, devido ao seu comportamento heterógeno, que se adapta as funcionalidades de hardware de diferentes modelos de comunicação, de acordo com os parâmetros de modulação/demodulação e codificação/decodificação definidos via software.

A implementação dos moduladores (transmissores), é o primeiro passo na construção do RDS, qual em sua configuração completa, integra as funcionalidades referentes aos demoduladores (receptores), permitindo com que a aplicação seja capaz de transmitir e receber sinais utilizando a melhor técnica de modulação para cada cenário enfrentado.

6.2 Projetos Futuros

Para a continuidade do projeto, têm-se as seguintes atividades:

- Depuração do erro de temporização: ajustar os moduladores digitais de forma a somente iniciarem a transmissão de um novo símbolo após completar o ciclo de onda referente ao último símbolo transmitido.
- Desenvolvimento de demoduladores digitais: construir o lado receptor do RDS, através da integração de blocos demoduladores que sejam capazes de receber o sinal modulado e identificar cada símbolo transmitido num dado instante de tempo, podendo assim passar adiante essa informação para uma outra aplicação.
- Desenvolvimento do conversor D/A: construir ou adquirir um módulo conversor que seja compatível com as configurações de hardware/software do projeto, a fim de visualizar os sinais modulados em Osciloscópio.
- Desenvolvimento antena de comunicação: construir antena capaz de transmitir e receber os sinais gerados pelo RDS, respeitando todos os seus parâmetros e de implementação e configuração.

REFERÊNCIAS BIBLIOGRÁFICAS.

[1] I F. AKYILDIZ et al., "Wireless Sensor Networks: A Survey" *Comp. Networks*, vol. 38, no. 4, Mar, 2002, pp. 393-422.

[2] AZZEDINE BOUKERCHE, HORACIO A. B. F. OLIVEIRA, EDUARDO F. NAKAMURA and ANTONIO A. F. LOUREIRO, "Localization Systems for Wireless Sensor Networks", *IEEE Wireless Communications*, 2007.

[3] P.G. BURNS, "Software Defined Radio for 3G", *Artech House – England*, 2003.

[4] "Software-Defined Radio", *Wipro Technologies*, 2002.

[5] GRÉGORY, E.N, M.S and François. V, "Transaction Level Modeling of SCA Compliant Software Defined Radio Waveforms and Platforms PIM/PSM", *Design, Automation & Test in Europe Conference & Exhibition*, 16-20 April 2007.

[6] B.P LATHI, ZHI DING. "Sistemas de Comunicações Analógicos e Digitais Modernos", LTC, Rio de Janeiro, Brasil, 4ª Edição, 2012.

[7] ALL ABOUT CIRCUITS – Understanding Frequency Modulation – Beginner. Disponível em: https://forum.allaboutcircuits.com/threads/understanding-frequency-modulation-beginner.126625/. Acesso em 14 de Junho de 2017.

[8] JEFFREY H. REED. "Software RADIO – A Modern Approach to Radio Engineering", Prentice Hall, New Jersey, 2002.

[9] AMÍLCAR CARELI CÉSAR, "Modulação Digital e Analógica – SEL 371 Sistemas de Comunicação", Departamento de Engenharia Elétrica da EESC-USP, 2006.

[10] MAGNA DESIGN NET, "Technical Notes – OFDM - Digital Modulation". Disponível em: http://www.magnadesignnet.com/en/booth/technote/ofdm/page2.php. Acesso em 17 de Junho de 2017.

[11] PROFESSOR MARLIO BONFIM. "Técnicas de Modulação – Modulação QAM". Disponível em: http://www.eletr.ufpr.br/marlio/te241/aula3.pdf. Acesso em 17 de Junho de 2017.

[12] ANALOG DEVICES "Fundamentals of Direct Digital Synthesis (DDS)".

[13] MATHWORKS "HDL Coder – Generate Verilog and VHDL code for FPGA and ASIC designs". Disponível em: https://www.mathworks.com/products/hdl-coder.html. Acesso em 18 de Junho de 2017.

[14] XILINX "Field Programmable Gate Array (FPGA)". Disponível em: https://www.xilinx.com/training/fpga/fpga-field-programmable-gate-array.htm. Acesso em 18 de Junho de 2017.

[15] DIGILENT "Nexys 3 FPGA Board Reference Manual", 2016.

[16] WAKERLY, J. F. "Digital Design – Principals and Practices", Prentice Hall, 2000.

[17] TUTTLEBEE, W. “Software Defined Radio: Enabling Technologies”, Wiley, 2002.

[18] XILINX “MicroBlaze Processor Reference Guide”, 2009.

[19] KENINGTON, P. B. “RF and Baseband Techniques for Software Defined Radio”, Artech House. 2005.

[20] SEDRA A., SMITH K. “Microeletrônica”. Prentice Hall, São Paulo, Brazil, 5ª Ed, 2013.

[21] PEDRONI A. VOLNEI. “Circuit Design with VHDL”. MIT Press. 2004.

[22] XILINX. “ISE Design Suite Overview”. Disponível em: https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ise_c_overview.htm. Acesso em 02/07/2017.

[23] XILINX. “ISim User Guide” Disponível em: https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/plugin_ism.pdf. Acesso em 02/07/2017.

[24] XILINX. “Vivado Design Suite User Guide”. Disponível em: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_3/ug910-vivado-getting-started.pdf

[25] IFSC, MEDEIROS DIEGO. “Predefinição: Diego Medeiros – SST20707”. Disponível em: https://wiki.sj.ifsc.edu.br/wiki/index.php/Predefini%C3%A7%C3%A3o:DiegoMedeiros-SST20707. Acesso em 10/12/2017.

[26] RASPBERRY PI. “Raspberry Pi 3 Model B - Specifications”. Disponível em: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/. Acesso em 10/12/2017.

APÊNDICES

APÊNDICE A – CÓDIGO VHDL MODULADOR 8-ASK

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity eigth_ask is
    Port ( data_in : in  STD_LOGIC_VECTOR (2 downto 0);
           amp_cos : out  STD_LOGIC_VECTOR (4 downto 0);
           amp_sin : out  STD_LOGIC_VECTOR (4 downto 0));
end eigth_ask;

architecture Behavioral of eigth_ask is

begin

        process (data_in)
        begin
        case data_in is
                when "000" => amp_sin <= "00000"; -- 0
                when "001" => amp_sin <= "00010"; -- 2
                when "010" => amp_sin <= "00100"; -- 4
                when "011" => amp_sin <= "00110"; -- 6
                when "100" => amp_sin <= "01000"; -- 8
                when "101" => amp_sin <= "01010"; -- 10
                when "110" => amp_sin <= "01100"; -- 12
                when "111" => amp_sin <= "01110"; -- 14
                when others => amp_sin <= "00000";
        end case;
        end process;

        amp_cos <= "00000";

end Behavioral;
```

APÊNDICE B – CÓDIGO VHDL BLOCO WAVE GENERATOR (10 AMOSTRAS)

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY teste_seno IS
  PORT( clk                         :   IN    std_logic;
        reset                       :   IN    std_logic;
        clk_enable                  :   IN    std_logic;
                  var_freq                      :   IN    std_logic_vector(2 DOWNTO 0); --
variavel de frequencia
```

```
        cos_wave                          :    OUT    std_logic_vector(15 DOWNTO 0);   --
sfix16_En14
        sin_wave                          :    OUT    std_logic_vector(15 DOWNTO 0)   --
sfix16_En14
        );/
END teste_seno;


ARCHITECTURE rtl OF teste_seno IS

  -- Signals
  SIGNAL enb                        : std_logic;
  SIGNAL Sine_Wave_out1_re              : signed(15 DOWNTO 0);  -- sfix16_En14
  SIGNAL Sine_Wave_out1_im              : signed(15 DOWNTO 0);  -- sfix16_En14
  SIGNAL address_cnt                : unsigned(3 DOWNTO 0);  -- ufix4
  SIGNAL sum_freq                                                          : integer;


BEGIN
  enb <= clk_enable;

-- DEF FREQ

  process(var_freq)
  begin
  case var_freq is
         when "000" => sum_freq <= 1;
         when "001" => sum_freq <= 2;
         when "010" => sum_freq <= 3;
         when "011" => sum_freq <= 4;
         when "100" => sum_freq <= 5;
         when "101" => sum_freq <= 6;
         when "110" => sum_freq <= 7;
         when others => sum_freq <= 8;
         end case;
  end process;


-- ADDRESS COUNTER
  Sine_Wave_addrcnt_temp_process1 : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      address_cnt <= to_unsigned(0, 4);
    ELSIF clk'event AND clk = '1' THEN
      IF enb = '1' THEN
        IF address_cnt = to_unsigned(9, 4) THEN
          address_cnt <= to_unsigned(0, 4);
        ELSE
          address_cnt <= address_cnt + sum_freq;
        END IF;
```

```
      END IF;
    END IF;
  END PROCESS Sine_Wave_addrcnt_temp_process1;

-- FULL WAVE LOOKUP TABLE
  PROCESS(address_cnt)
  BEGIN
    CASE address_cnt IS
      WHEN "0000" => Sine_Wave_out1_re <= "0100000000000000";
      WHEN "0001" => Sine_Wave_out1_re <= "0011001111000111";
      WHEN "0010" => Sine_Wave_out1_re <= "0001001111000111";
      WHEN "0011" => Sine_Wave_out1_re <= "1110110000111001";
      WHEN "0100" => Sine_Wave_out1_re <= "1100110000111001";
      WHEN "0101" => Sine_Wave_out1_re <= "1100000000000000";
      WHEN "0110" => Sine_Wave_out1_re <= "1100110000111001";
      WHEN "0111" => Sine_Wave_out1_re <= "1110110000111001";
      WHEN "1000" => Sine_Wave_out1_re <= "0001001111000111";
      WHEN "1001" => Sine_Wave_out1_re <= "0011001111000111";
      WHEN OTHERS => Sine_Wave_out1_re <= "0011001111000111";
    END CASE;

          cos_wave <= std_logic_vector(Sine_Wave_out1_re);

  END PROCESS;

  PROCESS(address_cnt)
  BEGIN
    CASE address_cnt IS
      WHEN "0000" => Sine_Wave_out1_im <= "0000000000000000";
      WHEN "0001" => Sine_Wave_out1_im <= "0010010110011110";
      WHEN "0010" => Sine_Wave_out1_im <= "0011110011011110";
      WHEN "0011" => Sine_Wave_out1_im <= "0011110011011110";
      WHEN "0100" => Sine_Wave_out1_im <= "0010010110011110";
      WHEN "0101" => Sine_Wave_out1_im <= "0000000000000000";
      WHEN "0110" => Sine_Wave_out1_im <= "1101101001100010";
      WHEN "0111" => Sine_Wave_out1_im <= "1100001100100010";
      WHEN "1000" => Sine_Wave_out1_im <= "1100001100100010";
      WHEN "1001" => Sine_Wave_out1_im <= "1101101001100010";
      WHEN OTHERS => Sine_Wave_out1_im <= "1101101001100010";
    END CASE;

          sin_wave <= std_logic_vector(Sine_Wave_out1_im);

  END PROCESS;

END rtl;
```

APÊNDICE C – CÓDIGO VHDL MODULADOR 8-PSK

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity eigth_psk is
    Port ( data_in : in  STD_LOGIC_VECTOR (2 downto 0);
           amp_cos : out  STD_LOGIC_VECTOR (4 downto 0);
           amp_sin : out  STD_LOGIC_VECTOR (4 downto 0));
end eigth_psk;

architecture Behavioral of eigth_psk is

begin

        process (data_in)
        begin
        case data_in is
                when "000" => amp_cos <= "00000"; amp_sin <= "01010"; -- 0, 10
                when "001" => amp_cos <= "00111"; amp_sin <= "00111"; -- 7, 7
                when "010" => amp_cos <= "01010"; amp_sin <= "00000"; -- 10, 0
                when "011" => amp_cos <= "00111"; amp_sin <= "11001"; -- 7, -7
                when "100" => amp_cos <= "00000"; amp_sin <= "10110"; -- 0, -10
                when "101" => amp_cos <= "11001"; amp_sin <= "11001"; -- -7, -7
                when "110" => amp_cos <= "10110"; amp_sin <= "00000"; -- -10, 0
                when "111" => amp_cos <= "11001"; amp_sin <= "00111"; -- -7, 7
                when others => amp_cos <= "00000"; amp_sin <= "00000";
        end case;
        end process;
end Behavioral;
```

APÊNDICE D – CÓDIGO VHDL MODULADOR 16-QAM

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE IEEE.numeric_std.ALL;

entity sixteen_qam is
    Port ( data_in : in  STD_LOGIC_VECTOR (3 downto 0);
           amp_cos : out STD_LOGIC_VECTOR (4 downto 0);
                     amp_sin : out STD_LOGIC_VECTOR (4 downto 0));
end sixteen_qam ;

architecture Behavioral of sixteen_qam is

begin

        process (data_in)
        begin
```

```
        case data_in is
                when "0000" => amp_cos <= "10001"; amp_sin <= "01111"; -- -15, 15
                when "0001" => amp_cos <= "10001"; amp_sin <= "00101"; -- -15, 5
                when "0010" => amp_cos <= "10001"; amp_sin <= "11011"; -- -15, -5
                when "0011" => amp_cos <= "10001"; amp_sin <= "10001"; -- -15, -15
                when "0100" => amp_cos <= "11011"; amp_sin <= "01111"; -- -5, 15
                when "0101" => amp_cos <= "11011"; amp_sin <= "00101"; -- -5, 5
                when "0110" => amp_cos <= "11011"; amp_sin <= "11011"; -- -5, -5
                when "0111" => amp_cos <= "11011"; amp_sin <= "10001"; -- -5, -15
                when "1000" => amp_cos <= "00101"; amp_sin <= "01111"; -- 5, 15
                when "1001" => amp_cos <= "00101"; amp_sin <= "00101"; -- 5, 5
                when "1010" => amp_cos <= "00101"; amp_sin <= "11011"; -- 5, -5
                when "1011" => amp_cos <= "00101"; amp_sin <= "10001"; -- 5, -15
                when "1100" => amp_cos <= "01111"; amp_sin <= "01111"; -- 15, 15
                when "1101" => amp_cos <= "01111"; amp_sin <= "00101"; -- 15, 5
                when "1110" => amp_cos <= "01111"; amp_sin <= "11011"; -- 15, -5
                when "1111" => amp_cos <= "01111"; amp_sin <= "10001"; -- 15, -15
                when others => amp_cos <= "00000"; amp_sin <= "00000";
        end case;
        end process;
end Behavioral;
```

APÊNDICE E – CÓDIGO VHDL MULTIPLICADOR

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

entity multiplicador is
    Port ( wave : in  STD_LOGIC_VECTOR (11 downto 0);
           amplitude : in  STD_LOGIC_VECTOR (4 downto 0);
           result : out  STD_LOGIC_VECTOR (16 downto 0));
end multiplicador;

architecture Behavioral of multiplicador is

begin

        result <= amplitude * wave;

end Behavioral;
```

APÊNDICE F – CÓDIGO VHDL SOMADOR

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
```

```
entity somador is
    Port ( X : in  STD_LOGIC_VECTOR (16 downto 0);
           Y : in  STD_LOGIC_VECTOR (16 downto 0);
           S : out  STD_LOGIC_VECTOR (16 downto 0));
end somador;

architecture Behavioral of somador is

begin

        S <= X + Y;

end Behavioral;
```

APÊNDICE G – CÓDIGO VHDL REGISTRADOR SIPO 4 BITS

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity sipo is
   Port ( clk : in  STD_LOGIC;
          clear : in  STD_LOGIC;
          input_data : in  STD_LOGIC;
          output_data : out  STD_LOGIC_VECTOR (3 downto 0));
end sipo;

architecture Behavioral of sipo is

begin

        sipo : process (clk,input_data,clear) is
        variable s : std_logic_vector(3 downto 0):= "0000";

        begin

                if (clear = '1') then
                        s := "0000";
                elsif (rising_edge(clk)) then
                        s:= (input_data & s(3 downto 1));
                end if;

                output_data <= s;

        end process sipo;

end Behavioral;
```

APÊNDICE H – CÓDIGO VHDL REGISTRADOR SIPO 3 BITS

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity sipo_three is
    Port ( clk : in  STD_LOGIC;
           clear : in  STD_LOGIC;
           input_data : in  STD_LOGIC;
           output_data : out  STD_LOGIC_VECTOR (2 downto 0));
end sipo_three;

architecture Behavioral of sipo_three is

begin

           sipo_three : process (clk,input_data,clear) is
           variable s : std_logic_vector(2 downto 0):= "000";

           begin

                    if (clear = '1') then
                              s := "000";
                    elsif (rising_edge(clk)) then
                              s:= (input_data & s(2 downto 1));
                    end if;

                    output_data <= s;

           end process sipo_three;

end Behavioral;
```

APÊNDICE I – CÓDIGO VHDL MULTIPLEXADOR

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity multiplex is
    Port ( qam_data : in  STD_LOGIC_VECTOR (16 downto 0);
           psk_data : in  STD_LOGIC_VECTOR (16 downto 0);
           ask_data : in  STD_LOGIC_VECTOR (16 downto 0);
                      fsk_data : in  STD_LOGIC_VECTOR (16 downto 0);
           sel : in  STD_LOGIC_VECTOR (1 downto 0);
           data_out : out  STD_LOGIC_VECTOR (16 downto 0));
end multiplex;

architecture Behavioral of multiplex is
```

```
begin

        process (sel, qam_data, psk_data, ask_data, fsk_data)
        begin
        case sel is
                when "00" => data_out <= qam_data;
                when "01" => data_out <= psk_data;
                when "10" => data_out <= ask_data;
                when "11" => data_out <= fsk_data;
                when others => data_out <= "0000000000000000";
        end case;
        end process;

end Behavioral;
```

APÊNDICE J – CÓDIGO VHDL RDS

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity rds_v1 is
    Port ( data_in : in  STD_LOGIC;
                           clk_rds : in  STD_LOGIC;
                           clk_register : in  STD_LOGIC;
                           clk_enable_rds : in STD_LOGIC;
                           reset_rds : in  STD_LOGIC;
                           clear_rds : in STD_LOGIC;
                           wave_sel : in STD_LOGIC_VECTOR(1 downto 0);
          data_modulate : out  STD_LOGIC_VECTOR (16 downto 0));
end rds_v1;

architecture Behavioral of rds_v1 is

-- Constantes
signal freq_v : std_logic_vector(2 downto 0);
signal amp_fsk : std_logic_vector(4 downto 0);

signal out_sipo_4 : std_logic_vector(3 downto 0);
signal out_sipo_3 : std_logic_vector(2 downto 0);
signal amp_sin_ask : std_logic_vector(4 downto 0);
signal amp_sin_psk : std_logic_vector(4 downto 0);
signal amp_cos_psk : std_logic_vector(4 downto 0);
signal amp_cos_ask : std_logic_vector(4 downto 0);
signal amp_cos_qam : std_logic_vector(4 downto 0);
signal amp_sin_qam : std_logic_vector(4 downto 0);
signal cos_qam : std_logic_vector(11 downto 0);
signal sin_qam : std_logic_vector(11 downto 0);
signal mult_cos_qam : std_logic_vector(16 downto 0);
signal mult_sin_qam : std_logic_vector(16 downto 0);
```

```
signal mult_cos_psk : std_logic_vector(16 downto 0);
signal mult_sin_psk : std_logic_vector(16 downto 0);
signal wave_qam : std_logic_vector(16 downto 0);
signal cos_ask : std_logic_vector(11 downto 0);
signal sin_ask : std_logic_vector(11 downto 0);
signal wave_ask : std_logic_vector(16 downto 0);
signal cos_psk : std_logic_vector(11 downto 0);
signal sin_psk : std_logic_vector(11 downto 0);
signal wave_psk : std_logic_vector(16 downto 0);
signal cos_fsk : std_logic_vector(11 downto 0);
signal sin_fsk : std_logic_vector(11 downto 0);
signal wave_fsk : std_logic_vector(16 downto 0);


component eigth_ask is
   Port ( data_in : in  STD_LOGIC_VECTOR (2 downto 0);
          amp_cos : out  STD_LOGIC_VECTOR (4 downto 0);
          amp_sin : out  STD_LOGIC_VECTOR (4 downto 0));
end component;

component eigth_psk is
   Port ( data_in : in  STD_LOGIC_VECTOR (2 downto 0);
          amp_cos : out  STD_LOGIC_VECTOR (4 downto 0);
          amp_sin : out  STD_LOGIC_VECTOR (4 downto 0));
end component;

component multiplex is
   Port ( qam_data : in  STD_LOGIC_VECTOR (16 downto 0);
          psk_data : in  STD_LOGIC_VECTOR (16 downto 0);
          ask_data : in  STD_LOGIC_VECTOR (16 downto 0);
                       fsk_data : in  STD_LOGIC_VECTOR (16 downto 0);
          sel : in  STD_LOGIC_VECTOR (1 downto 0);
          data_out : out  STD_LOGIC_VECTOR (16 downto 0));
end component;

component multiplicador is
   Port ( wave : in  STD_LOGIC_VECTOR (11 downto 0);
          amplitude : in  STD_LOGIC_VECTOR (4 downto 0);
          result : out  STD_LOGIC_VECTOR (16 downto 0));
end component;

component sipo_three is
   Port ( clk : in  STD_LOGIC;
          clear : in  STD_LOGIC;
          input_data : in  STD_LOGIC;
          output_data : out  STD_LOGIC_VECTOR (2 downto 0));
end component;

component sipo is
   Port ( clk : in  STD_LOGIC;
```

```
        clear : in  STD_LOGIC;
        input_data : in  STD_LOGIC;
        output_data : out  STD_LOGIC_VECTOR (3 downto 0));
end component;

component sixteen_qam is
    Port ( data_in : in  STD_LOGIC_VECTOR (3 downto 0);
        amp_cos : out STD_LOGIC_VECTOR (4 downto 0);
                    amp_sin : out STD_LOGIC_VECTOR (4 downto 0));
end component;

component somador is
    Port ( X : in  STD_LOGIC_VECTOR (16 downto 0);
        Y : in  STD_LOGIC_VECTOR (16 downto 0);
        S : out  STD_LOGIC_VECTOR (16 downto 0));
end component;

component waves_12bits IS
  PORT( clk                     :   IN    std_logic;
       reset                    :   IN    std_logic;
       clk_enable               :   IN    std_logic;
       var_freq                 :   IN    std_logic_vector(2 DOWNTO 0); --variavel de
frequencia
       cos_wave                 :   OUT   std_logic_vector(11 DOWNTO 0);  --
sfix12_En10
       sin_wave                 :   OUT   std_logic_vector(11 DOWNTO 0)  --
sfix12_En10
       );
END component;


begin

        freq_v <= "000";
        amp_fsk <= "00001";

entrada1: sipo port map (
                    clk => clk_register,
                    clear => clear_rds,
                    input_data => data_in,
                    output_data => out_sipo_4);

entrada2: sipo_three port map (
                    clk => clk_register,
                    clear => clear_rds,
                    input_data => data_in,
                    output_data => out_sipo_3);

entrada3: eigth_ask port map (
                    data_in => out_sipo_3,
```

```
                amp_cos => amp_cos_ask,
                amp_sin => amp_sin_ask);

entrada4: eigth_psk port map (
                data_in => out_sipo_3,
                amp_cos => amp_cos_psk,
                amp_sin => amp_sin_psk);

entrada5: sixteen_qam port map (
                data_in => out_sipo_4,
                amp_cos => amp_cos_qam,
                amp_sin => amp_sin_qam);

entrada6: waves_12bits port map (
                clk => clk_rds,
                reset => reset_rds,
                clk_enable => clk_enable_rds,
                var_freq => freq_v,
                cos_wave => cos_qam,
                sin_wave => sin_qam);

entrada7: multiplicador port map (
                wave => cos_qam,
                amplitude => amp_cos_qam,
                result => mult_cos_qam);

entrada8: multiplicador port map (
                wave => sin_qam,
                amplitude => amp_sin_qam,
                result => mult_sin_qam);

entrada9: somador port map (
                X => mult_cos_qam,
                Y => mult_sin_qam,
                S => wave_qam);

entrada10: waves_12bits port map (
                clk => clk_rds,
                reset => reset_rds,
                clk_enable => clk_enable_rds,
                var_freq => freq_v,
                cos_wave => cos_ask,
                sin_wave => sin_ask);

entrada11: multiplicador port map (
                wave => sin_ask,
                amplitude => amp_sin_ask,
                result => wave_ask);

entrada12: waves_12bits port map (
```

```
                clk => clk_rds,
                reset => reset_rds,
                clk_enable => clk_enable_rds,
                var_freq => freq_v,
                cos_wave => cos_psk,
                sin_wave => sin_psk);

entrada13: multiplicador port map (
                wave => sin_psk,
                amplitude => amp_sin_psk,
                result => mult_sin_psk);

entrada14: multiplicador port map (
                wave => cos_psk,
                amplitude => amp_cos_psk,
                result => mult_cos_psk);

entrada15: somador port map (
                X => mult_cos_psk,
                Y => mult_sin_psk,
                S => wave_psk);

entrada16: waves_12bits port map (
                clk => clk_rds,
                reset => reset_rds,
                clk_enable => clk_enable_rds,
                var_freq => out_sipo_3,
                cos_wave => cos_fsk,
                sin_wave => sin_fsk);

entrada17: multiplicador port map (
                wave => sin_fsk,
                amplitude => amp_fsk,
                result => wave_fsk);

entrada18: multiplex port map (
                qam_data => wave_qam,
                psk_data => wave_psk,
                ask_data => wave_ask,
                fsk_data => wave_fsk,
                sel => wave_sel,
                data_out => data_modulate);

end Behavioral;
```

APÊNDICE K – CÓDIGO VHDL TEST BENCH RDS

```
library IEEE;
use IEEE.Std_logic_1164.all;
```

```
use IEEE.Numeric_Std.all;

entity rds_v1_tb is
end;

architecture bench of rds_v1_tb is

  component rds_v1
     Port ( data_in : in  STD_LOGIC;
                          clk_rds : in  STD_LOGIC;
                          clk_register : in  STD_LOGIC;
                          clk_enable_rds : in STD_LOGIC;
                          reset_rds : in  STD_LOGIC;
                          clear_rds : in STD_LOGIC;
                          wave_sel : in STD_LOGIC_VECTOR(1 downto 0);
            data_modulate : out  STD_LOGIC_VECTOR (16 downto 0));
  end component;

  signal data_in: STD_LOGIC := '0';
  signal clk_rds: STD_LOGIC := '0';
  signal clk_register: STD_LOGIC := '0';
  signal clk_enable_rds: STD_LOGIC := '0';
  signal reset_rds: STD_LOGIC := '0';
  signal clear_rds: STD_LOGIC := '0';
  signal wave_sel: STD_LOGIC_VECTOR(1 downto 0) := "00";
  signal data_modulate: STD_LOGIC_VECTOR (16 downto 0) := (others=>'0');

begin

  uut: rds_v1 port map ( data_in        => data_in,
                       clk_rds        => clk_rds,
                       clk_register   => clk_register,
                       clk_enable_rds => clk_enable_rds,
                       reset_rds      => reset_rds,
                       clear_rds      => clear_rds,
                       wave_sel       => wave_sel,
                       data_modulate  => data_modulate );

         clk_rds <= not clk_rds after 0.5 us;
         clk_register <= not clk_register after 25 us;
         reset_rds <= '0', '1' after 0.5 ns, '0' after 1 ns, '1' after 0.5 ns, '0' after 1 ns;
         data_in <= not data_in after 50 us;
         clk_enable_rds <= '1';
         clear_rds <= '0';
         wave_sel <= "00";
end;
```