MINISTÉRIO DA CIÊNCIA E TECNOLOGIA
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

# IMPLEMENTAÇÃO DE UM DECODIFICADOR SBCDA/ARGOS EM FPGA

Bárbara Silva de Souza (UFRN, Bolsista PIBIC/CNPq)
E-mail: barbara.souza@crn.inpe.br

José Marcelo Lima Duarte (INPE-CRN, Orientador)
E-mail: jmarcelo@crn.inpe.br

Natal, Rio Grande do Norte.
Julho de 2017

MINISTÉRIO DA CIÊNCIA E TECNOLOGIA
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

# IMPLEMENTAÇÃO DE UM DECODIFICADOR SBCDA/ARGOS EM FPGA

RELATÓRIO FINAL DE PROJETO DE INICIAÇÃO CIENTÍFICA
(PIBIC/CNPq/INPE)

Bárbara Silva de Souza (UFRN, Bolsista PIBIC/CNPq)
E-mail: barbara.souza@crn.inpe.br

José Marcelo Lima Duarte (INPE-CRN, Orientador)
E-mail: jmarcelo@crn.inpe.br

Natal, Rio Grande do Norte.
Julho de 2017

# Resumo

Esse documento contém o relato do trabalho e estudos desenvolvidos durante o período da bolsa de iniciação cientifica no INPE-CRN (Centro Regional do Nordeste). O objetivo principal deste trabalho foi converter blocos de PDS do modelo MatLab do decodificador desenvolvido para o Sistema Brasileiro de Coleta de Dados Ambientais a nível RTL. O conteúdo deste relatório tem como objetivo apresentar a funcionalidade e descrição dos códigos dos blocos PDS a nível RTL e os resultados obtidos em comparação ao do modelo em MatLab.

SUMÁRIO

# LISTA DE FIGURAS

# INTRODUÇÃO

No ano de 1993, foi iniciada a operação do Sistema Brasileiro de Coleta de Dados Ambientais (SBCDA). Essa operação foi iniciada devido à grande necessidade de um sistema automático de aquisição de dados ambientais e do desenvolvimento de tecnologia espacial nacional.

O SBCDA conta com a aquisição de dados ambientais a partir de plataformas de coletas de dados (PCDs) espalhadas pelo território nacional. Devido à extensão do país, surge a necessidade de transmissão de dados via satélite, devido a localização de algumas PCDs.

O sistema de comunicação atual do SBCDA funciona como um "espelho". O satélite retransmite o sinal das PCDs sem decodificá-los para as Estações Terrenas de Recepção (ETRs). Apesar de simples de embarcar, o sistema atual é passível de falhas, tais como: gasto de energia desnecessário do transmissor, que tem precisa estar ligado o tempo inteiro e também a perda de dados durante a transmissão entre as PCDs e o satélite.

Com a finalidade de decodificar e armazenar os dados enviados das PCDs diretamente no satélite faz-se necessário projetar um decodificador para embarcar no satélite. É neste contexto que o presente projeto está inserido. Mais especificamente, no projeto de implementação em FPGA (*Field Programmable Gate Array*) de um decodificador para o sinal do SBCDA (*Sistema Brasileiro de Coleta de Dados Ambientais*).

O objetivo desse trabalho é contribuir no desenvolvimento do firmware do FPGA, realizando a conversão de parte do modelo do algoritmo de decodificação, já desenvolvido em MATLAB, em um modelo RTL (*Register Transfer Level*). Mais especificamente, realizando a conversão do bloco de demodulação do sinal.

Neste sentido, foram feitos códigos HDL (*Hardware Description Language*) para implementar o modelo RTL de algumas funções de processamento digital de sinais que são utilizadas pelo demodulador, bem como a integração destas funções compondo o próprio demodulador. Estes códigos foram validados a partir de *testbenches*, que são códigos usados na verificação do design implementado. A estratégia de validação utilizada compara as saídas obtidas do modelo RTL com as do modelo em MATLAB para uma mesma entrada.

Esse relatório contém a descrição da funcionalidade do demodulador e dos blocos que o compõem, além da descrição dos códigos HDL traduzidos e de como os testes foram feitos. Os códigos propriamente ditos se encontram no anexo.

## DEMODULADOR

O sinal transmitido por uma PTT sofre variações de frequência durante seu percurso. A função do demodulador em questão é realizar um sincronismo de frequência eliminando o ajuste de fase da portadora pura deslocando o sinal na frequência para zero, restando apenas o sinal em banda base.

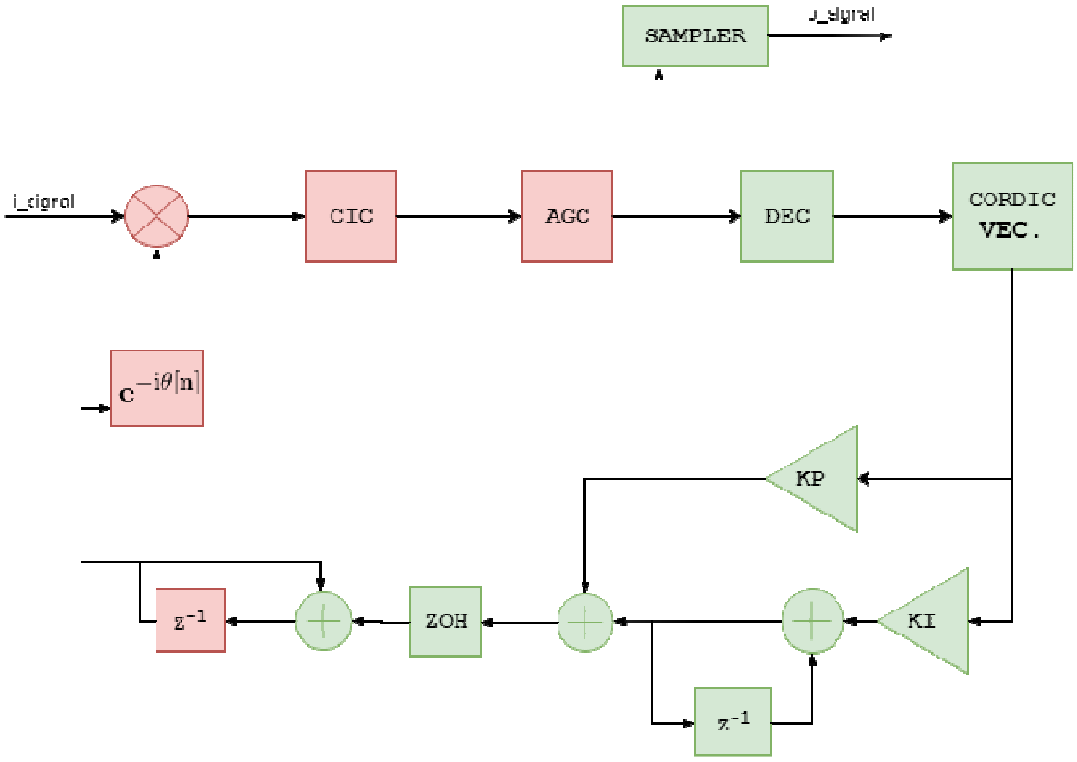


**Figura 1: Diagrama de blocos do demodulador**

O demodulador é composto por um NCO (numerically controled oscillator), um multiplicador complexo, um filtro casado, um AGC (automatic gain control), um detector de fase e um loop filter. O filtro casado foi implemetado com um filtro CIC decimador de estágio único. Já o detector de fase foi implemetado por um algoritmo CORDIC no modo *vectoring*. Na Figura 1 pode-se observar o diagrama de blocos do demodulador.

O sinal de entrada e a saída do NCO são multiplicados para gerar um sinal em banda base. O resultado dessa multiplicação é enviado para o filtro casado, de modo a maximizar a SNR. O *forward* AGC ajusta a saída do filtro para caber em uma variável de 10 bits. Essa amplitude ajustada é o sinal de saída do sistema, que é entrada do bloco sampler, que realiza o sincronismo de símbolo. Para fechar o PLL (phased locked loop), a saída do AGC passa por uma decimação e pelo loop filter para gerar o controle de frequência do sinal do NCO.

Em mais detalhes, o sistema funciona da seguinte forma:

- 1280 amostras do sinal de entrada são enviadas para o bloco de decodificação, assim como a frequência da PTT detectada;
- A frequência recebida é utilizada como frequência inicial desse sistema, que gera um sinal cuja fase é igual a da portadora utilizando-o para demodular essas 1280 amostras recebidas;
- O sinal agora com frequência próxima a zero, é filtrado e decimado pelo filtro CIC, eliminando assim possíveis sinais de PTTs vizinhas e melhorando a SNR.
- A saída do filtro é normalizada pelo AGC e decimada mais uma vez.
- A fase do sinal que sai do bloco AGC é extraída utilizando o CORDIC no modo vectoring, e enviada para o Loop Filter.
- A saída do loop filter é enviada para o bloco NCO, que gera uma frequência de ajuste que será somada a frequência utilizada anteriormente para demodular o sinal.
- Para o novo sinal demodulado o mesmo processo se repete.

A saída do bloco demodulador conta com o sinal saindo do AGC e passando pelo *sampler*. Esse bloco é utilizado para realizar o sincronismo de símbolo. Para um caso em que o sincronismo de símbolo foi perfeitamente atingido, estaríamos coletando as amostras no tempo correto.

Se houver erro no tempo de coleta das amostras, estaríamos coletando pontos próximos, mas com amplitudes menores devido à interferência intersimbólica. Para corrigir o erro, o estimador de tempo envia o erro de tempo para um filtro interpolador, que interpola a amostra recebida de modo a encontrar uma estimativa da amostra sem interferência intersimbólica.

## LOOP FILTER

Um *phase-locked loop* (PLL) é um sistema de controle que gera um sinal de saída com frequência e fase instantâneas em sincronismo com o sinal de entrada. Por manter em regime permanente a frequência de saída igual à frequência de entrada, tal sistema pode rastrear a frequência do sinal amostrado ou gerar frequências múltiplas do sinal de entrada.

Tais propriedades desse controlador permitem seu uso em aplicações como análise de sinais, telecomunicações, controle de sistemas elétricos de potência entre outros.

Para este trabalho, um PLL é utilizado para realizar a demodulação do sinal. O PLL básico é composto por um detector de fase, um *Loop Filter* e um oscilador. No capítulo anterior, temos a composição do modulador, ou seja, o PLL utilizado.

O projeto do *Loop Filter* precisa estar de acordo com as especificações do PLL, que são:

- **Tempo de estabilização ≤ 60 ms:** Já que o sincronismo deve ser atingido durante o período de portadora, o qual tem duração de 160 ms. O sincronismo de frequência deve ser também lento o suficiente para que não sofra fortes influências do ruído.
- Criticamente amortecido;
- Máxima aceleração *doppler* do sinal de entrada = 120 Hz/s;
- A banda passante deve ser a mínima que atenda os critérios de tempo de estabilização e erro em regime.

O projeto do PLL foi feito durante a implementação do modelo em MatLab. A função de transferência obtida para o *Loop Filter* foi a seguinte:

$$\boldsymbol{H(z) = \frac{K_p + K_i - z^{-k_p}}{1 - z^{-1}}}$$

Sendo $\boldsymbol{K_p = 1.04e^{-3}}$ e $\boldsymbol{K_i = 4.34e^{-5}}$. Para implementação do modelo RTL, foi necessário arredondar os valores dessas constantes, tornando-as números inteiros e realizando a operação de *shift* para se obter novamente o valor.

## AGC

O AGC estima um valor médio absoluto do sinal de entrada, usando uma média móvel e aplicando um ganho variável para ajustar esse valor médio para o nível desejado. Isso reduz a faixa dinâmica do sinal, fazendo com o que o sinal possa ser representado com menos bits.

## NCO

Um *numerically controlled oscillator* (NCO) é um gerador de sinal digital que cria uma forma de onda síncrona e discreta, normalmente uma senoide. São muito utilizados em sistemas de comunicações, como PLLs digitais, sistemas de radares, transmissão acústica e óptica entre outros.

Um NCO consiste em uma *look-up table* (LUT) contendo dados da forma de onda e um acumulador do índice que varre a LUT. O raio de mudança do contador determina a frequência da onda de saída, em unidade normalizada.

O NCO utilizado neste projeto gera o seguinte valor:

$$\boldsymbol{Y = M * e^{2i\pi Index/N}}$$

Em que M = 1, *Index* é o endereço para acessar na memória ROM contendo valores dados por

$$\boldsymbol{round\left(M * e^{\frac{2i\pi\left[1:\frac{N}{8}\right]}{N}}\right)}$$

Em que $\boldsymbol{N = 2^{ADDR\_W}}$ sendo ADDR_W o número de bits do endereço de memória.

# FILTRO CIC

O filtro CIC – “Cascade Integrator-Comb” é uma implementação computacional eficiente de filtros passa-baixa que são frequentemente embarcados em implementações de decimação e interpolação em hardware para sistemas de comunicação modernos. A principal vantagem do filtro CIC, comparado a outros filtros digitais, é que ele é implementado utilizando apenas somas e subtrações, de modo a reduzir significativamente a carga computacional.

A função do filtro CIC de primeira ordem é a filtragem do sinal depois de ser demodulado. Com o sinal tendo frequências próximas a zeros, é usado de modo a eliminar possíveis frequências das PCDs vizinhas, mantendo apenas a portadora do sinal recebido.

O filtro CIC é considerado o aprimoramento de um filtro média móvel não recursivo, mostrado na imagem a seguir:

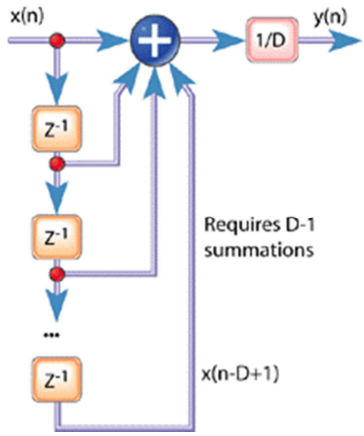


**Figura 2: Filtro Média Móvel Não Recursiva**

Nota-se que o filtro requer D-1 somas e uma divisão por D; A saída desse filtro é dada pela seguinte expressão no tempo:

$$y(n) = \frac{1}{D}[x(n) + x(n-1) + x(n-2) + \cdots + x(n-D+1)]$$

Pode-se implementar um filtro média móvel recursivo substituindo $y(n-1)\ em\ y(n)$, obtendo:

$$y(n) = \frac{1}{D}[x(n) - x(n-D)] + \ y(n-1)]$$

A equação acima corresponde ao filtro média móvel da figura abaixo:

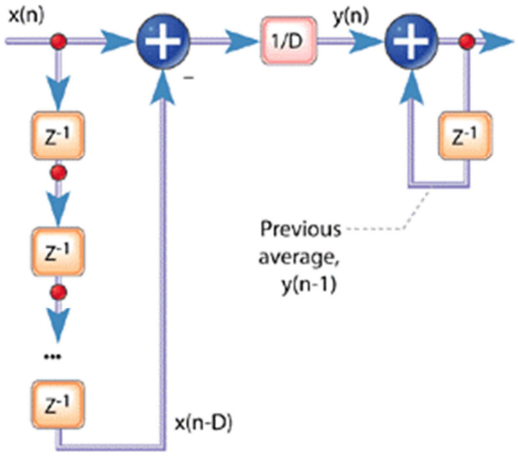


**Figura 3: Filtro Média Móvel Recursiva**

A estrutura do filtro CIC consiste em condensar a representação do atraso e ignorar a divisão por D, obtendo assim a forma clássica o filtro CIC de primeira ordem, o qual a estrutura em cascata é mostrada na figura a seguir:

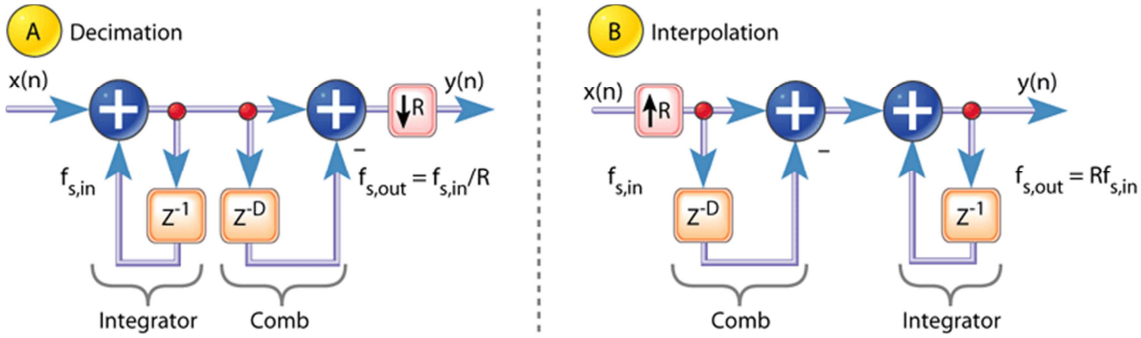


**Figura 4: Filtro CIC**

A parte da realimentação positiva do filtro é chamada de seção comb (pente), o qual possui um atraso diferencial D, enquanto que a seção de realimentação negativa é chamada integrator (integrador). A seção comb subtrai o atraso da atual amostra de entrada e o integrator é simplesmente um acumulador. A equação do CIC é dada por:

$$y(n) = [x(n) - x(n - D)] + y(n - 1)$$

O bloco PDS, que foi convertido, trata-se do filtro CIC no modo de operação de decimação, cujo objetivo é reduzir a taxa de amostragem sinal a ser filtrado. Sua estrutura é mostrada na figura a seguir:

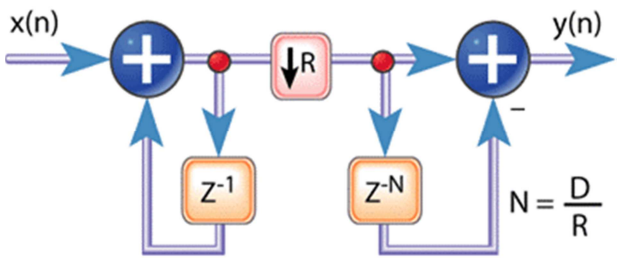


**Figura 5: Filtro CIC de primeira ordem - Decimação**

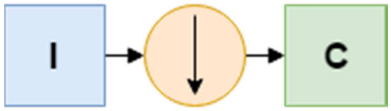


**Figura 6: Estrutura do filtro CIC**

A estrutura final é formada pela associação do bloco integrador, do parâmetro de decimação R e do comb, necessariamente nessa ordem. Como é visto na Figura 6.

O caso mais básico da decimação consiste em não levar em conta algumas amostras, diminuindo assim a frequência de amostragem. O parâmetro “R” trata-se da taxa de redução da amostragem. Uma decimação por dois significa que R é igual a dois e, numa decimação básica, a cada duas amostras uma será desconsiderada. No caso de uma decimação por quatro, três amostras serão ignoradas e assim sucessivamente.

O código RTL foi descrito de modo a permitir a alteração da taxa de amostragem assim como do atraso diferencial, para permitir que um mesmo bloco pudesse ser usado na composição de outros sem alterar o código.

## CORDIC ITERATIVO NO MODO VECTORING

O CORDIC – “Coordinate Rotation Digital Computer” é um método com uma sequência iterativa de adições/subtrações e operações de deslocamento, que representam rotações através de ângulos de rotações pré-definidos, mas com direção de rotação variável, usado para calcular funções trigonométricas, lineares ou hiperbólicas usando componentes de hardware mínimos como deslocadores lógicos (shift) e comparadores.

### *Equações Básicas do CORDIC*

O algoritmo do CORDIC, que opera sobre sucessivas rotações de vetores num plano cartesiano, é derivado da transformada geral de rotação:

$$x' = xcos\emptyset - ysin\emptyset$$

$$y' = ycos\emptyset + xsin\emptyset$$

O qual rotaciona o vetor no plano cartesiano pelo ângulo ϕ. Que pode ser rearranjado da seguinte maneira:

$$x' = cos\emptyset[x - ytan\emptyset]$$

$$y' = cos\emptyset[y + xsin\emptyset]$$

Se os ângulos de rotação forem restritos a $\tan\emptyset = \pm 2^{i}$, de maneira que a multiplicação pelo termo da tangente é reduzida a uma simples operação de shift. Ângulos arbitrários de rotação podem ser obtidos ao realizar uma série de pequenas rotações. Se a variável *i* em cada iteração é a direção de rotação ao invés de ser uma variável para definir se há rotação ou não, então o termo $cos\emptyset$ vira uma constante. Logo a expressão de rotação iterativa pode ser expressa por:

$$x_{i+1} = K_i\left[x_i - y_i d_i 2^{-i}\right]$$

$$y_{i+1} = K_i[y_i + x_i d_i 2^{-i}]$$

Onde:

$$K_i = 1/\sqrt{1 + 2^{-2i}}$$

$$d_i = \pm 1$$

O termo $K_i$ refere-se ao ganho do CORDIC. A simplificação não influência o ângulo de rotação, mas os pontos resultantes são afetados, de maneira que o módulo do vetor não representa o raio da circunferência do vetor rotacionado. Esse ganho converge para aproximadamente 0,607 e sua função é recuperar os valores originais dos vetores, tal como uma constante de normalização.

Para saber por quais ângulos elementares o processo já foi executado implementa-se um acumulador de ângulo, que é uma terceira equação adicionada ao algoritmo do CORDIC.

$$z_{i+1} = z_i - d_i \tan^{-1}(2^{-i})$$

O CORDIC possui dois modos de operação:

- modo Rotação (*Rotation*), onde as coordenadas de um vetor e o ângulo de rotação são dados e são calculadas as coordenadas do novo vetor após a rotação do ângulo dado;

- modo Vetorização (*Vectoring*), onde as coordenadas de um vetor são dadas e é calculado magnitude do vetor original e o ângulo de rotação resultante.

No modo vetorização, que foi o utilizado nesse projeto, o CORDIC rotaciona o vetor de entrada através de qualquer ângulo necessário para alinhar o vetor resultante ao eixo x. A função vetorização funciona de modo a minimizar a componente y do vetor residual a cada rotação. O índice da componente y residual é usado para determinar qual a próxima direção de rotação. As equações para o modo vetorização são:

$$x_{i+1} = x_i - y_i d_i 2^{-i}$$

$$y_{i+1} = y_i + x_i d_i 2^{-i}$$

$$z_{i+1} = z_i - d_i \tan^{-1}(2^{-i})$$

Onde $d_i = +1$ se $y_i < 0$, e -1 caso contrário, o que provê os seguintes resultados:

$$x_n = A_n \sqrt{x_0^2 + y_0^2}$$

$$y_n = 0$$

$$z_n = z_0 + \tan^{-1} \frac{y_0}{x_0}$$

$$A_n = \prod_n \sqrt{1 + 2^{-2i}}$$

O CORDIC rotação ou vetorização têm ângulos de rotação limitados entre $\pm 2\pi$. Para compor ângulos de rotação maior que essa limitação, faz-se necessário usar uma rotação adicional de $\pm \pi/2$. A correção iterativa é dada por:

$$x' = -d * y$$

$$y' = d * x$$

$$z' = z + d * \frac{\pi}{2}$$

Onde $d = +1$ se $y < 0$, e -1 caso contrário.

# CODIFICAÇÃO EM VERILOG E VALIDAÇÃO

Para realizar a codificação e compilação dos códigos feitos, foi utilizado no software ModelSim.

A estrutura dos códigos HDL referentes aos blocos citados no capítulo anterior possuem um padrão básico, contendo: entradas e saídas de dados, entrada de *clock* e *reset*, e as entradas de controle, tais como: *enable*, *input_valid* e *output_valid*, e algumas outras dependendo de sua finalidade. Esse padrão foi elaborado devido à necessidade de integração destes blocos posteriormente. Os códigos referentes ao modelo RTL do demodulador se encontram no capítulo *Apêndice*.

A entrada *enable* é a que define o funcionamento de cada módulo, ou seja, caso essa entrada esteja desabilitada o bloco se mantém inativo. Por essa razão, para os testes realizados, essa entrada permanece em 1 sempre.

A entrada *input_valid* é a que permite a entrada de dados para cada módulo, ou seja, quando essa entrada receber 1, será iniciado o recebimento de dados imediatamente após 1 pulso de clock.

A Figura 7 contém um exemplo genérico da forma de onda de cada bloco.

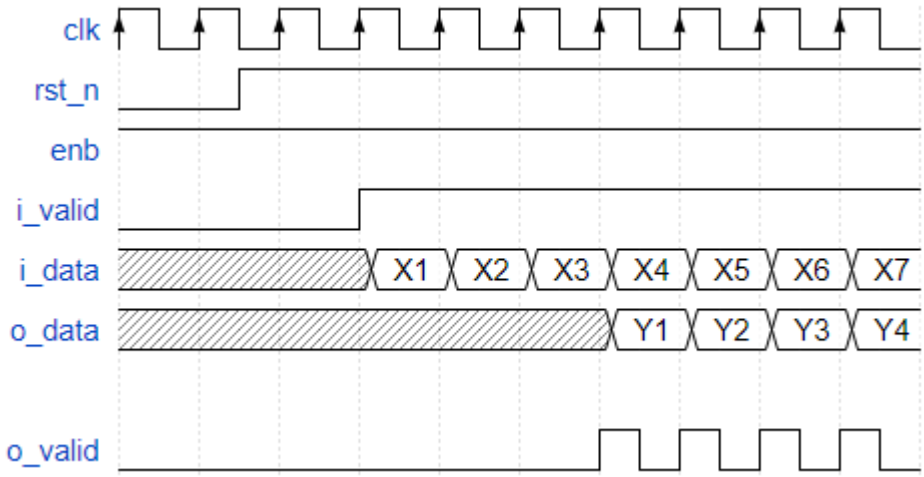


**Figura 7: Forma de onda**

É importante salientar que cada bloco possui um tempo de processamento dos dados de saída diferente. O sinal de controle *output_valid* vai aguardar pelo menos um pulso de clock após a habilitação do sinal de entrada *input_valid* para então ser habilitado. E só então é que teremos os dados de saída do bloco.

Para realização dos testes de cada módulo foi necessário acrescentar ao modelo em MATLAB um código que armazena os vetores com dados de entrada e saída, individualmente, em um arquivo *txt*.

Para testar os módulos foi criado um *testbench*, que é um código, descrito na linguagem SystemVerilog, usado para injetar um sinal de entrada e analisar a saída de cada módulo. O código do testbench padrão é dividido entre partes estruturadas que funcionam paralelamente.

Primeiro cria-se a instância do módulo a ser testado, o *Design Under Test* (DUT), essa instância associa as entradas e saídas do módulo com as do *testbench*; a outra parte recebe o sinal de entrada, no caso o arquivo *txt* com o vetor de dados de entrada do modelo em MATLAB. A última parte faz a comparação entre a saída do RTL com o vetor de dados de saída do MATLAB e retorna uma mensagem de erro, caso não forem iguais, e de sucesso, caso sejam iguais.

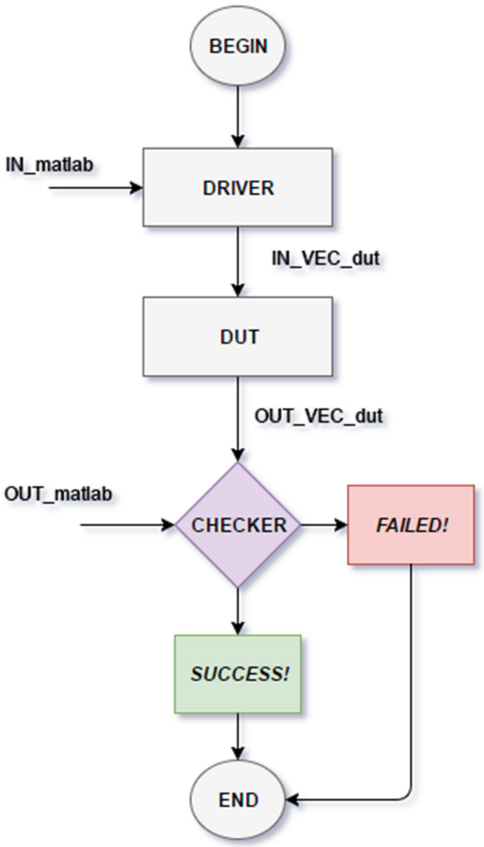


**Figura 8: Diagrama de blocos do *testbench***

O *Driver* lê o arquivo do MatLab e o traduz em um estímulo de entrada para o DUT; o *Checker* captura o sinal de saída do DUT e o converte em um vetor de dados para ser comparado com a saída esperada do MatLab. Caso a saída do RTL seja igual a do MatLab, é imprimido na tela a mensagem “Success!”, caso não seja igual, a mensagem que aparece aparecerá é “Failed!” e junto a ela o valor obtido diante do valor esperado.

## RESULTADOS OBTIDOS

A Figura 9 contém a janela com as mensagens de saída que indicam com a situação da simulação.

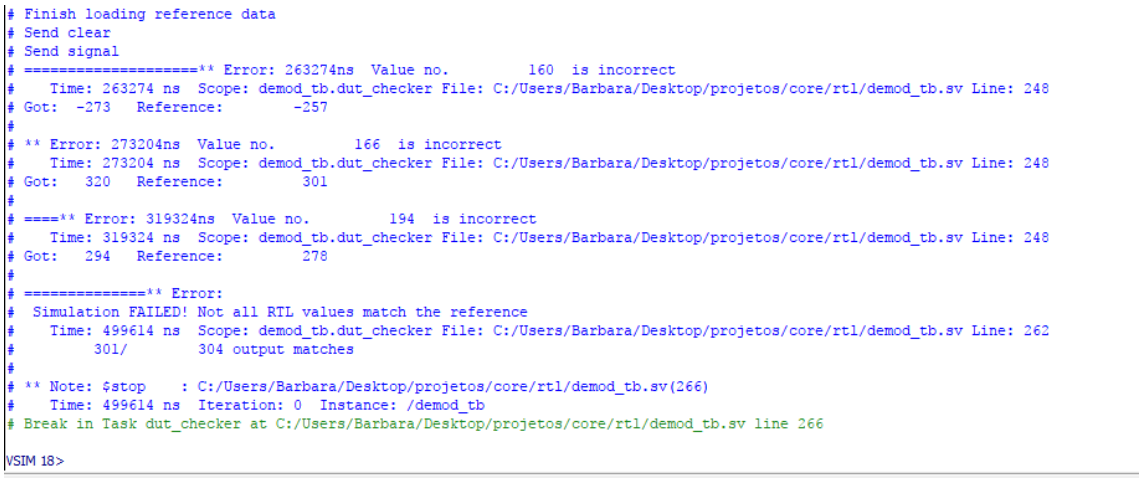


**Figura 9: Janela com mensagens da situação da simulação**

É possível ver que das 304 saídas obtidas, apenas três não tiveram valores exatamente iguais as do modelo MatLab. Apesar disso, esse resultado foi considerado muito bom. À medida que o projeto vai se desenvolvendo, possíveis correções serão realizadas.

# CONCLUSÃO

O trabalho desenvolvido até o presente momento mostra que o sistema atende as especificações e é compatível com o modelo desenvolvido em MatLab. Os próximos passos para o projeto consistem em integrar o demodulador ao sistema do decodificador e implementar o decodificador no FPGA, realizando testes com sinais reais, mais próximos da utilização real do decodificador.

# REFERÊNCIAS

ANDRAKA, Ray. A survey of CORDIC algorithms for FPGA based computers. Andraka Consulting Group, Inc.

LYONS, R. G. (2011). Understanding Digital Signal Processing (3ª Edição). Prentice Hall.

VAHID, Frank. Digital Design: with RTL, VHDL and Verilog (2ª Edição).

ASIC WORLD. Tutorial de linguagem Verilog. Disponível em: < http://www.asic-world.com>. Acesso em 19 de julho de 2017.

EMBEDDED. Understanding cascaded integrator-comb filters. Disponível em: < http://www.embedded.com/design/configurable-systems/4006446/Understanding-cascaded-integrator-comb-filters>. Citado no capítulo *Filtro CIC*, fazendo uso das figuras. Acesso em 19 de julho de 2017.

# APÊNDICE

## CÓDIGO FILTRO CIC

```
`timescale 1ns/100ps
module cic_filter #(
  parameter WIDTH_IN  = 16,
  parameter WIDTH_OUT = 24, // = WIDTH_IN+ceil(log2(DECIRATE*DIFF_DLY))
  parameter DECIRATE  = 40,
  parameter DIFF_DLY  = 4 ,
  parameter COUNT_W   = 6   // ceil(log2(DECIRATE))
) (
  //input wire i_valid;
  input                          clk    ,
  input                          rst_n  ,
  input                          enb    ,
  input                          clear  ,
  input                          i_valid,
  input      signed [ WIDTH_IN-1:0] i_x_re ,
  input      signed [ WIDTH_IN-1:0] i_x_im ,
  output reg                        o_valid,
  output reg signed [WIDTH_OUT-1:0] o_y_re ,
  output reg signed [WIDTH_OUT-1:0] o_y_im
);

// integrator stage register
  reg signed [WIDTH_OUT-1:0] acc_re   ;
  reg signed [WIDTH_OUT-1:0] acc_im   ;
  reg                        acc_valid;

// Decimation counter
  reg  [COUNT_W-1:0] count     ;
  reg                last_count;

// Shift Register
  reg signed [WIDTH_OUT-1:0] shftreg_re[0:DIFF_DLY-1]; //buffer
  reg signed [WIDTH_OUT-1:0] shftreg_im[0:DIFF_DLY-1];

// comb
wire signed [WIDTH_OUT-1:0] comb_re;
wire signed [WIDTH_OUT-1:0] comb_im;

//-------------------------------------------------------------------------
// sample valid propagation logic
//-------------------------------------------------------------------------

always @(posedge clk, negedge rst_n) begin
  if(!rst_n) begin
    o_valid <= 1'b0;
    acc_valid <= 1'b0;
  end else if (clear || enb) begin
    // acculator output generates one output one cycle after last_count;
    acc_valid <= !clear & (i_valid & last_count);
    o_valid   <= acc_valid;
  end
end
```

```
//-------------------------------------------------------------------------
// counter logic
//-------------------------------------------------------------------------

  reg [COUNT_W-1:0] next_count;
  wire count_enb;

  assign count_enb = clear || (enb && i_valid);

  always @(*) begin
    last_count = count >= DECIRATE-1;
    if (clear || last_count) begin
      next_count = {COUNT_W{1'b0}};
    end else begin
      next_count = count + 1;
    end
  end

  always @(posedge clk, negedge rst_n) begin
    if(!rst_n) begin
      count  <= {COUNT_W{1'b0}};
    end else if (count_enb) begin
      count <= next_count;
    end
  end
//-------------------------------------------------------------------------
// integrate logic
//-------------------------------------------------------------------------

reg signed [WIDTH_OUT-1:0] next_acc_re;
reg signed [WIDTH_OUT-1:0] next_acc_im;

always @(*) begin : acc_comb_u
    if (clear) begin
      next_acc_re = {WIDTH_OUT{1'b0}};
      next_acc_im = {WIDTH_OUT{1'b0}};
    end else begin
      next_acc_re = i_x_re + acc_re;
      next_acc_im = i_x_im + acc_im;
    end
  end

always @(posedge clk, negedge rst_n) begin : acc_reg_u
  if(!rst_n) begin
    acc_re    <= {WIDTH_OUT{1'b0}};
    acc_im    <= {WIDTH_OUT{1'b0}};
  end else if (count_enb) begin
    acc_re <= next_acc_re;
    acc_im <= next_acc_im;
  end
end
```

```
//------------------------------------------------------------------------
// shift register, comb, and output register logic
//------------------------------------------------------------------------

wire shft_enb;
integer i0;

assign shft_enb = clear || (enb && acc_valid);

always @(posedge clk, negedge rst_n) begin : shft_reg_u
  if(!rst_n) begin
    for (i0=0; i0<DIFF_DLY; i0=i0+1) begin
      shftreg_re[i0] <= {WIDTH_OUT{1'b0}};
      shftreg_im[i0] <= {WIDTH_OUT{1'b0}};
    end
  end else if (shft_enb) begin
    shftreg_re[0] <= acc_re;
    shftreg_im[0] <= acc_im;
    for (i0=0; i0<DIFF_DLY-1; i0=i0+1) begin
      shftreg_re[i0+1] <= shftreg_re[i0];
      shftreg_im[i0+1] <= shftreg_im[i0];
    end
  end
end


//------------------------------------------------------------------------
// Comb and output register logic
//------------------------------------------------------------------------

assign comb_re = acc_re - shftreg_re[DIFF_DLY-1];
assign comb_im = acc_im - shftreg_im[DIFF_DLY-1];

always @(posedge clk, negedge rst_n) begin : out_reg_u
  if(!rst_n) begin
    o_y_re <= {WIDTH_OUT{1'b0}};
    o_y_im <= {WIDTH_OUT{1'b0}};
  end else if (shft_enb) begin
    o_y_re <= comb_re;
    o_y_im <= comb_im;
  end
end

endmodule
```

## CÓDIGO CORDIC VECTORING ITERATIVE

```
module cordic_it #(
  parameter DATA_W  = 16,
  parameter NUM_OF_IT  = 8,
  parameter II_W  = 4,      // ceil(log2(NUM_OF_IT))
  parameter GAIN_ADJ = 1'b1,
  parameter K_MANT = 155,
  parameter GAIN_FRAC = 8
) (
  input clk,
  input rst_n,
  input enb,
  input sink_valid,
  input vec_mod,
  input signed [DATA_W-1:0] i_x,
  input signed [DATA_W-1:0] i_y,
  input signed [DATA_W-1:0] i_theta,
  input source_ready,
  output sink_ready,
  output reg signed [DATA_W-1:0] o_x,
  output reg signed [DATA_W-1:0] o_y,
  output reg signed [DATA_W-1:0] o_theta,
  output reg source_valid
);

  //----------------Internal Variables--------------------------------//
  // from FSM for sink_ready generation
  localparam [1:0]
    WAIT_INPUT=2'b10,
      BUSY=2'b00,
        WAIT_READ=2'b01;
  reg [1:0] state;  // sink_ready=state[1];

  // from initial rotation
  reg signed [DATA_W-1:0] x_rot;
  reg signed [DATA_W-1:0] y_rot;
  reg signed [DATA_W-1:0] theta_rot;

  // from mux
  reg signed [DATA_W-1:0] x_mux;
  reg signed [DATA_W-1:0] y_mux;
  reg signed [DATA_W-1:0] theta_mux;

  // from output/feedback register
  reg signed [DATA_W-1:0] x_reg;
  reg signed [DATA_W-1:0] y_reg;
  reg signed [DATA_W-1:0] theta_reg;

  // from counter
  wire count_enb;
  wire count_last;
  reg [II_W-1:0]  ii;
```

```
// from aritimetic unit
  reg signed [DATA_W-1:0] x_n;
  reg signed [DATA_W-1:0] y_n;
  reg signed [DATA_W-1:0] theta_n;
  reg signed [DATA_W-1:0] d_x;
  reg signed [DATA_W-1:0] d_y;
  reg signed [DATA_W-1:0] x_aux;
  reg signed [DATA_W-1:0] y_aux;
  reg signed source_reg;
  //reg signed [DATA_W-1:0] theta_dly;
  wire signed [31:0] rom_data0 [0:15];
  wire signed [DATA_W-1:0] rom_out;

  //-------------------------------------------------------------------------//
  //-----------------Assignments output--------------------------------------//

  // assign o_x = x_aux;
  // assign o_y = y_aux;
  // assign o_theta = theta_aux;

  // output register
  always @(posedge clk, negedge rst_n) begin
    if (!rst_n) begin
      o_x <= 0;
      o_y <= 0;
      o_theta <= 0;
      source_valid <= 0;
    end
    else if (enb) begin
      if (source_reg) begin
        o_x <= x_aux;
        o_y <= y_aux;
        o_theta <= theta_reg;
      end
      source_valid <= source_reg;
    end
  end

  generate
    if(GAIN_ADJ == 1) begin
      always @(*) begin
        x_aux = (x_reg*K_MANT)>>>GAIN_FRAC;
        y_aux = (y_reg*K_MANT)>>>GAIN_FRAC;
      end
    end else begin
      always @(*) begin
        x_aux = x_reg;
        y_aux = y_reg;
      end
    end
  endgenerate

  //-----------------ROM-----------------------------------------------------//
  assign rom_data0[0]  = 32'b00100000000000000000000000000000;//0
  assign rom_data0[1]  = 32'b00001001011100100000001010001110;//-1
  assign rom_data0[2]  = 32'b00000100111111011001110000101011;//-2
  assign rom_data0[3]  = 32'b00000010100010001000100010111010100;//-3
  assign rom_data0[4]  = 32'b00000001010001011000011010100011;//-4
  assign rom_data0[5]  = 32'b00000000101000101110101111100001;//-5
  assign rom_data0[6]  = 32'b00000000010100010111011000011110;//-6
  assign rom_data0[7]  = 32'b00000000001010001011111000101010;//-7
  assign rom_data0[8]  = 32'b00000000000101000101111001010011;//-8
  assign rom_data0[9]  = 32'b00000000000010100010111110010111;//-9
  assign rom_data0[10] = 32'b00000000000001010001011111001100;//-10
  assign rom_data0[11] = 32'b00000000000000101000101111001100;//-11
  assign rom_data0[12] = 32'b00000000000000010100010111110011;//-12
  assign rom_data0[13] = 32'b00000000000000001010001011111001;//-13
  assign rom_data0[14] = 32'b00000000000000000101000101111010;//-14
  assign rom_data0[15] = 32'b00000000000000000010100010111101;//-15
```

```
assign rom_out = rom_data0[ii][31:32-DATA_W] + rom_data0[ii][32-DATA_W-1];

//----------------------------------------------------------------------//
always @(*) begin
  //initial rotation (-pi/2 or pi/2)
  if(
    (vec_mod && i_y[DATA_W-1])||(!vec_mod && ^i_theta[DATA_W-1:DATA_W-2])) //
   begin

    x_rot = -i_y;
    y_rot = i_x;
    theta_rot = i_theta - {1'b1, {DATA_W-1{1'b0}}}/2; //dividir por 2 pra ser 90
  end else begin

    x_rot = i_y;
    y_rot = -i_x;
    theta_rot = i_theta + {1'b1, {DATA_W-1{1'b0}}}/2;
  end
end

// MUX between input and feedback paths
always @(*) begin : mux_u
  if(sink_ready) begin
    x_mux = x_rot;
    y_mux = y_rot;
    theta_mux = theta_rot;
  end else begin
    x_mux = x_reg;
    y_mux = y_reg;
    theta_mux = theta_reg;
  end
end

// cordic aritimetic unit
always @ (*)
  begin : aritimetic_u
    d_x = x_mux >>> ii;
    d_y = y_mux >>> ii;
    if ((!vec_mod && theta_mux[DATA_W-1] == 0)||
        (vec_mod && y_mux[DATA_W-1])) begin
      x_n = x_mux - d_y;
      y_n = y_mux + d_x;
      theta_n = theta_mux - rom_out;
    end else begin
      x_n = x_mux + d_y;
      y_n = y_mux - d_x;
      theta_n = theta_mux + rom_out;
    end
  end

// counter enable logic
assign count_enb = (sink_valid && sink_ready) || ii!=0;
// last count value flag
assign count_last = ii >= NUM_OF_IT-1;

// counter
always @(posedge clk, negedge rst_n) begin : counter_u
  if(!rst_n) begin
    ii <= {II_W{1'b0}};
  end else if(enb) begin
    if(count_enb) begin
      if(count_last) begin
        ii <= {II_W{1'b0}};
      end else begin
        ii <= ii + 1;
      end
    end
  end
end
```

```
// output/feedback register

  always @ (posedge clk, negedge rst_n)
    begin: outreg_u
      if (!rst_n) begin
        x_reg <= 0;
        y_reg <= 0;
        theta_reg <= 0;
        source_reg <=1'b0;
      end
      else if(enb) begin
        x_reg <= x_n;
        y_reg <= y_n;
        theta_reg <= theta_n;
        source_reg <= count_last;
      end
    end
  // FSM for sink_ready generation

  assign sink_ready = state==WAIT_INPUT;

  always @(posedge clk, negedge rst_n) begin : fsm_u
    if(!rst_n) begin
      state <= WAIT_INPUT;
    end else if(enb) begin
      case (state)
        WAIT_INPUT:
          if (sink_valid) state <= BUSY;
        BUSY:
          if (count_last) begin
            if(source_ready) begin
              state <= WAIT_INPUT;
            end else begin
              state <= WAIT_READ;
            end
          end
        WAIT_READ:
          if (source_ready) state <= WAIT_INPUT;
        default:
          state <= WAIT_INPUT;
      endcase
    end
  end
endmodule
```

## CÓDIGO LOOP FILTER

```
module loop_filter #(
  parameter DIN_W = 12,
  parameter DOUT_PREC = 20,
  // frequency desviation track limit in Hz 2^(DOUT_W-DOUT_PREC-1)*128e3
  parameter DOUT_W = 12,
  // COEF_W is the required word width for a signed variable to store KP and KI constants
  parameter COEF_W = 8,
  parameter KP = 68, // Kp = KP * 2^(-KP_FRAC)
  parameter KI = 91, // Ki = KI * 2^(-KI_FRAC)
  parameter KP_FRAC = 16,
  parameter KI_FRAC = 21
) (
  input rst_n,
  input clk,
  input enb,
  input clear,
  input i_valid,
  input signed [DIN_W-1:0] din,
  output o_valid,
  output signed [DOUT_W-1:0] dout
);

localparam ACC_PREC = KI_FRAC + DIN_W;
localparam ACC_W = ACC_PREC - DOUT_PREC + DOUT_W;
localparam P_I_PREC = KP_FRAC + DIN_W;
localparam P_I_W = P_I_PREC - DOUT_PREC + DOUT_W;

reg signed [ACC_W-1:0] acc;
reg signed [ACC_W:0] acc_add;
wire signed [ACC_W-1:0] acc_sat;
reg signed [P_I_W:0] p_i_add;
reg signed [DOUT_W:0] p_i_trunc;

assign o_valid = i_valid;

always @(*) begin
  acc_add = acc + KI*din;
  p_i_add = (KP*din) + (acc >>> (KI_FRAC-KP_FRAC));
  p_i_trunc = p_i_add>>>(P_I_PREC-DOUT_PREC);
end

s_limit #(.WIDTH_IN(ACC_W+1), .WIDTH_OUT(ACC_W))
acc_limit_u (.din(acc_add), .dout(acc_sat));

s_limit #(.WIDTH_IN(DOUT_W+1), .WIDTH_OUT(DOUT_W))
p_i_limit_u (.din(p_i_trunc), .dout(dout));

always @(posedge clk, negedge rst_n) begin
  if(!rst_n || clear) begin
    acc <= {ACC_W{1'b0}};
  end else if(enb) begin
    if(i_valid) begin
      acc <= acc_sat;
    end
  end
end

endmodule
```

## CÓDIGO MULTIPLICADOR COMPLEXO

```
module cplx_mult
#(
  parameter X1_W=16,
  parameter X2_W=32,
  parameter Y_W=32,
  parameter ROUND_W=16,
  parameter ROUND = 0
)
(
  input clk,
  input rst_n,
  input signed [X1_W-1:0] x1_re,
  input signed [X1_W-1:0] x1_im,
  input signed [X2_W-1:0] x2_re,
  input signed [X2_W-1:0] x2_im,
  input enb,
  input signed i_valid,
  output reg signed o_valid,
  output reg signed [Y_W-1:0] y_re,
  output reg signed [Y_W-1:0] y_im
);

localparam Y0_W = X1_W+X2_W+1;
reg signed [Y0_W-1:0] y0_re, y0_im;
reg signed [Y0_W-ROUND_W-1:0] round_re, round_im;

always @(*) begin
  y0_re = x1_re*x2_re - x1_im*x2_im;
  y0_im = x1_re*x2_im + x1_im*x2_re;

  if(ROUND==0) begin
    round_re = y0_re >>> ROUND_W;
    round_im = y0_im >>> ROUND_W;
  end
  else begin
    round_re = y0_re[Y0_W-1: ROUND_W] +
      {{Y0_W-ROUND_W-1{1'b0}}, y0_re[ROUND_W-1]};

    round_im = y0_im[Y0_W-1:ROUND_W] +
      {{Y0_W-ROUND_W-1{1'b0}}, y0_im[ROUND_W-1]};
  end
        // y_re = round_re;
        // y_im = round_im;
end //always

always @(posedge clk or negedge rst_n) begin
  if(!rst_n) begin
    y_re    <= 0;
    y_im    <= 0;
    o_valid <= 1'b0;
  end else  if(enb) begin
    if(i_valid) begin
      y_re      <= round_re;
      y_im      <= round_im;
    end
    o_valid <= i_valid;
  end
end

endmodule
```

## CÓDIGO DO NCO

```
module nco
#(
  parameter CARTESIAN_W = 14,
  parameter FREQ_W = 21,
  parameter THETA_W = 11,
  parameter ONE = 2**(CARTESIAN_W-1)-1
)
(
  input clk,
  input rst_n,
  input enb,
  input sink_valid,            // Enable
  input [FREQ_W-1:0] freq_in,   // Frquency input
  input [THETA_W-1:0] theta_in, // Angle input
  output source_valid,
  output signed [CARTESIAN_W-1:0] nco_re,  // real value output
  output signed [CARTESIAN_W-1:0] nco_im   // imaginary value output
);

reg [FREQ_W-1:0] theta_acc;   // angle accumulator output
wire [FREQ_W-1:0] theta_rs;   // theta input resized to FREQ_W
wire [FREQ_W-1:0] theta_tot;  // total angle. Result from sum of theta_acc and theta_rs
wire [THETA_W-1:0] theta_tot_rs; // total angle resized to THETA_W
wire nco_tbl_valid;
wire signed [CARTESIAN_W-1:0] nco_tbl_re; // NCO Look-up-table output
wire signed [CARTESIAN_W-1:0] nco_tbl_im; // NCO Look-up-table output

// angle accumulator
always @(posedge clk, negedge rst_n) begin : theta_acc_u
  if(!rst_n) begin
    theta_acc <= {FREQ_W{1'b0}};
  end else if (enb) begin
    if(sink_valid) begin
      theta_acc <= theta_acc + freq_in;
    end
  end
end

// resize theta for FREQ_W, and sum it whith theta_acc
assign theta_rs = {theta_in, {FREQ_W-THETA_W{1'b0}}};
assign theta_tot = theta_acc + theta_rs;

// resize angle accumulator to THETA_W rouding LSBs
assign theta_tot_rs = (theta_tot>>(FREQ_W-THETA_W)) + {{THETA_W-1{1'b0}}, theta_tot[FREQ_W-THETA_W-1]};
```

```
// fetch cartesian value from LUT
nco_table
#(
  .CARTESIAN_W(CARTESIAN_W),
  .ADDR_W(THETA_W),
  .ONE(ONE),
  .OUTPUT_REG(1)
) nco_table_u0
(
  .clk(clk),
  .rst_n(rst_n),
  .enb(enb),
  .sink_valid(sink_valid),    // Enable
  .index(theta_tot_rs),       // Address input
  .source_valid(nco_tbl_valid),
  .dout_re(nco_tbl_re),
  .dout_im(nco_tbl_im)
);
assign source_valid = nco_tbl_valid;
assign nco_re = nco_tbl_re;
assign nco_im = nco_tbl_im;

endmodule
```

## CÓDIGO DO AGC

```
module forward_agc #(
  parameter DIN_W = 24,      // input word width
  parameter DIN_W_W = 5,     // ceil(log2(DIN_W+1))
  parameter DOUT_W = 10,     // output word width
  parameter MAX_MEAN_ABS = 2**19,
  parameter MIN_MEAN_ABS = 2**14,
  parameter LUT_ADDR_W= 5,   // mantissa gain look-up-table
  parameter VGA_FRAC = 7,    // mantissa gain LUT word width -1 (sign bit)
  // Average filter y(n) = y(n-1)*(1-2^-APLHA)+ 2^-ALPHA*x(n)
  parameter ALPHA_SHIFT = 5,
  parameter X_MSB_MAX = 20,  // Maximum expected MSB from input mean abs
  parameter X_MSB_MIN = 15   // Minimum expected MSB from input mean abs
) (
            input clk,    // Clock
            input enb,    // Clock Enable
            input rst_n,  // Asynchronous reset active low
  input clear,
            input signed [DIN_W-1:0] i_data_re,
            input signed [DIN_W-1:0] i_data_im,
            input i_valid,
            output reg signed [DOUT_W-1:0] o_data_re,
            output reg signed [DOUT_W-1:0] o_data_im,
            output [DIN_W-1:0] o_mean_abs,
            output reg o_valid
);

// instantaneos abs signals
reg signed [DIN_W-1:0] abs_re;
reg signed [DIN_W-1:0] abs_im;
reg signed [DIN_W-1:0] max;
reg signed [DIN_W-1:0] min;
reg signed [DIN_W:0] abs;
reg abs_valid;

// mean abs signals
wire signed [DIN_W:0] mean_abs;
wire [DIN_W-1:0] mean_abs_usgn;
wire mean_abs_valid;

// mean abs limited
reg [DIN_W-1:0] m_abs_limit;

// vga_exp signals
reg [DIN_W_W-1:0] x_msb;
reg [DIN_W_W-1:0] vga_exp;

// vga mantissa signals
localparam MANT_W = VGA_FRAC+2; //+1 for the integer part, and +1 for the sign
localparam LUT_L = 2**LUT_ADDR_W;
reg [LUT_ADDR_W:0] addr0;
reg [LUT_ADDR_W-1:0] addr;
reg signed [MANT_W-1:0] vga_mant;
wire signed [MANT_W-1:0] vga_lut [0:LUT_L-1];

// after mantissa gain
reg vga0_valid;
reg signed [DIN_W+VGA_FRAC-1:0] vga0_re;
reg signed [DIN_W+VGA_FRAC-1:0] vga0_im;
```

```
// after left shift
localparam MIN_SHIFT = X_MSB_MIN-DOUT_W-1;
localparam VGA_W = DIN_W+VGA_FRAC-MIN_SHIFT;
reg signed [VGA_W-1:0] vga_re;
reg signed [VGA_W-1:0] vga_im;

// after amplitude limit
wire signed [DOUT_W-1:0] limit_re;
wire signed [DOUT_W-1:0] limit_im;

// vga LUT
assign vga_lut[0]=128;
assign vga_lut[1]=124;
assign vga_lut[2]=120;
assign vga_lut[3]=117;
assign vga_lut[4]=114;
assign vga_lut[5]=111;
assign vga_lut[6]=108;
assign vga_lut[7]=105;
assign vga_lut[8]=102;
assign vga_lut[9]=100;
assign vga_lut[10]=98;
assign vga_lut[11]=95;
assign vga_lut[12]=93;
assign vga_lut[13]=91;
assign vga_lut[14]=89;
assign vga_lut[15]=87;
assign vga_lut[16]=85;
assign vga_lut[17]=84;
assign vga_lut[18]=82;
assign vga_lut[19]=80;
assign vga_lut[20]=79;
assign vga_lut[21]=77;
assign vga_lut[22]=76;
assign vga_lut[23]=74;
assign vga_lut[24]=73;
assign vga_lut[25]=72;
assign vga_lut[26]=71;
assign vga_lut[27]=69;
assign vga_lut[28]=68;
assign vga_lut[29]=67;
assign vga_lut[30]=66;
assign vga_lut[31]=65;


// instantaneous abs aproximation with max-min
always @(*) begin
  abs_re = (i_data_re<0) ? -i_data_re : i_data_re;
  abs_im = (i_data_im<0) ? -i_data_im : i_data_im;
  if (abs_re > abs_im) begin
    max = abs_re;
    min = abs_im;
  end else begin
    max = abs_im;
    min = abs_re;
  end
end

always @(posedge clk, negedge rst_n) begin : proc_
  if(~rst_n) begin
     abs_valid <= {DIN_W{1'b0}};
  end else if (enb) begin
     abs_valid <= i_valid;
     if (i_valid) begin
      abs <= max + (min >> 2);
     end
  end
end
```

```
// mean abs

lp_iir_1st_ord #(
  .DATA_W(DIN_W+1),
  .M(5))
mean_abs_u (
  .rst_n(rst_n),
  .clk(clk),
  .enb(enb),
  .clear(clear),
  .sink_valid(abs_valid),
  .din(abs),
  .source_valid(mean_abs_valid),
  .dout(mean_abs)
);
assign mean_abs_usgn = mean_abs;
assign o_mean_abs = mean_abs_usgn;

// limit mean_abs
always @(*) begin : mabs_limit_u
  if(mean_abs_usgn > MAX_MEAN_ABS) begin
    m_abs_limit = MAX_MEAN_ABS;
  end else if (mean_abs_usgn < MIN_MEAN_ABS) begin
    m_abs_limit = MIN_MEAN_ABS;
  end else begin
    m_abs_limit = mean_abs_usgn;
  end
end

// compute vga_exp
integer i0;
always @(*) begin : vga_exp_u
  x_msb = X_MSB_MIN;
  for (i0=X_MSB_MIN; i0<=X_MSB_MAX; i0=i0+1) begin
    if(m_abs_limit[i0]==1'b1) begin
      x_msb = i0+1;
    end
  end
  vga_exp = x_msb - (DOUT_W-1);
end

// compute vga_mant
always @(*) begin: vga_man_u
  addr0 = m_abs_limit >> (x_msb-LUT_ADDR_W-1);
  addr = addr0[LUT_ADDR_W-1:0];
  vga_mant = vga_lut[addr];
end

// apply mantissa gain
always @(posedge clk, negedge rst_n) begin
  if(!rst_n) begin
    vga0_re <= 0;
    vga0_im <= 0;
    vga0_valid <= 1'b0;
  end else if(enb) begin
    vga0_re <= vga_mant * i_data_re;
    vga0_im <= vga_mant * i_data_im;
    vga0_valid <= mean_abs_valid;
  end
end

// apply exp gain
always @(*) begin
  vga_re = (vga0_re >>> (vga_exp+VGA_FRAC)) +
    $signed({{DIN_W-1{1'b0}}, vga0_re[vga_exp+VGA_FRAC-1]});
  vga_im = (vga0_im >>> (vga_exp+VGA_FRAC)) +
    $signed({{DIN_W-1{1'b0}}, vga0_im[vga_exp+VGA_FRAC-1]});
end
```

```
// apply limit
s_limit #(.WIDTH_IN(VGA_W), .WIDTH_OUT(DOUT_W))
s_limit_u0 (.din(vga_re), .dout(limit_re));

s_limit #(.WIDTH_IN(VGA_W), .WIDTH_OUT(DOUT_W))
s_limit_u1 (.din(vga_im), .dout(limit_im));

// output register
always @(posedge clk, negedge rst_n) begin
  if(!rst_n) begin
    o_data_re <= 0;
    o_data_im <= 0;
    o_valid <= 1'b0;
  end else if(enb) begin
    o_data_re <= limit_re;
    o_data_im <= limit_im;
    o_valid <= vga0_valid;
  end
end

endmodule
```

## CÓDIGO DO DEMODULADOR

```
module demod #(
  parameter DIN_W = 16,
  parameter AGC_W = 10,
  parameter FREQ_W = 20,
  parameter THETA_W = 11,
  parameter MEAN_W = DIN_W+8,
  parameter NCO_W = 12,
  parameter AGC_DELAY = 32

) (
  input clk,
  input rst_n,
  input signed [DIN_W-1:0] i_signal_re,
  input signed [DIN_W-1:0] i_signal_im,
  input signed [FREQ_W-1:0] i_freq,
  input enb,
  input i_valid,
  input clear,
  output o_valid,
  output signed [AGC_W-1:0] o_symbl,
  output signed [MEAN_W-1:0] o_mean_abs,
  output [FREQ_W-1:0] o_freq,
  output o_symb_lock
);

//internal variables

// shift delay
 localparam NCO_DLY = 2;
 integer i0;
 reg signed [DIN_W-1:0] signal_dly_re [0:NCO_DLY-1];
 reg signed [DIN_W-1:0] signal_dly_im [0:NCO_DLY-1];

 // NCO
  wire signed [NCO_W-1:0] nco_re;
  wire signed [NCO_W-1:0] nco_im;
  wire nco_valid;

// complex mult
  wire signed [DIN_W-1:0] cplx_mult_re;
  wire signed [DIN_W-1:0] cplx_mult_im;
  wire cplx_mult_valid;

//cic filter
  localparam CIC_W = DIN_W+8; // 24
  wire signed [CIC_W-1:0] cic_re;
  wire signed [CIC_W-1:0] cic_im;
  wire cic_valid;

//agc
  wire [CIC_W-1:0] mean_abs;
  wire signed [AGC_W-1:0] agc_re;
  wire signed [AGC_W-1:0] agc_im;
  wire agc_valid;

//sampler
  wire signed [AGC_W-1:0] symbl;
  wire symbl_valid;
```

```
//cic for decimation
  localparam DEC_W = AGC_W+2; // 12
  wire signed [DEC_W-1:0] dec_re;
  wire signed [DEC_W-1:0] dec_im;
  wire dec_valid;

//cordic_vec
  // localparam CORDIC_W = DEC_W + 1; // para cordic paralelo
  localparam CORDIC_W = DEC_W + 2; // para cordic iterativo
  wire cordic_ready;
  wire signed [CORDIC_W-1:0] dec_rs_re;
  wire signed [CORDIC_W-1:0] dec_rs_im;
  wire signed [CORDIC_W-1:0] cordic_x;
  wire signed [CORDIC_W-1:0] cordic_y;
  wire signed [CORDIC_W-1:0] cordic_theta;
  wire cordic_valid;

//loop_filter
  localparam LPF_W = 12;
  wire lpf_valid;
  wire signed [LPF_W-1:0] lpf;

// freq desviation register
reg signed [LPF_W-1:0] dfreq;

 //-----------------------------------------------------------------------------
 // Shift Delay
 // - Synchronize input signal with NCO output
 //-----------------------------------------------------------------------------
always @(posedge clk or negedge rst_n) begin : nco_dly
  if(~rst_n) begin
    for (i0=0; i0<NCO_DLY; i0=i0+1) begin
      signal_dly_re[i0] <= {DIN_W{1'b0}};
      signal_dly_im[i0] <= {DIN_W{1'b0}};
    end
  end else if (enb) begin
    signal_dly_re[0] <= i_signal_re;
    signal_dly_im[0] <= i_signal_im;
    for (i0=1; i0<NCO_DLY; i0=i0+1) begin
      signal_dly_re[i0] <= signal_dly_re[i0-1];
      signal_dly_im[i0] <= signal_dly_im[i0-1];
    end
  end
end

nco #(
    .CARTESIAN_W(NCO_W  ),
    .FREQ_W     (FREQ_W ),
    .THETA_W    (THETA_W),
    .ONE(2**(NCO_W-1)-1)
  ) nco_u (
    .clk        (clk           ),
    .rst_n      (rst_n         ),
    .enb        (enb           ),
    .sink_valid  (i_valid       ),
    .freq_in     (o_freq        ),
    .theta_in    ({THETA_W{1'b0}}),
    .source_valid(nco_valid     ),
    .nco_re      (nco_re        ),
    .nco_im      (nco_im        )
  );
```

```
cplx_mult #(
  .X1_W   (DIN_W ),
  .X2_W   (NCO_W ),
  .Y_W    (DIN_W ),
  .ROUND_W(NCO_W-1),
  .ROUND(0)
) cplx_mult_u (
  .clk    (clk          ),
  .rst_n  (rst_n        ),
  .x1_re  (signal_dly_re[NCO_DLY-1]),
  .x1_im  (signal_dly_im[NCO_DLY-1]),
  .x2_re  (nco_re       ),
  .x2_im  (nco_im       ),
  .enb    (enb          ),
  .i_valid(nco_valid    ),
  .o_valid(cplx_mult_valid),
  .y_re   (cplx_mult_re ),
  .y_im   (cplx_mult_im )
);

cic_filter #(
  .WIDTH_IN (DIN_W),
  .WIDTH_OUT(CIC_W),
  .DECIRATE (40  ),
  .DIFF_DLY (4   ),
  .COUNT_W  (6   )
) mf_u (
  .clk    (clk          ),
  .rst_n  (rst_n        ),
  .enb    (enb          ),
  .clear  (clear        ),
  .i_x_re (cplx_mult_re ),
  .i_x_im (cplx_mult_im ),
  .i_valid(cplx_mult_valid),
  .o_valid(cic_valid    ),
  .o_y_re (cic_re       ),
  .o_y_im (cic_im       )
);

forward_agc #(
  .DIN_W (CIC_W),
  .DOUT_W(AGC_W),
  .DIN_W_W(5),
  .MAX_MEAN_ABS(2**19),
  .MIN_MEAN_ABS(2**14),
  .LUT_ADDR_W(5),
  .VGA_FRAC(7),
  .ALPHA_SHIFT(5),
  .X_MSB_MAX(20),
  .X_MSB_MIN(15)
) forward_agc_u (
  .clk      (clk    ),
  .enb      (enb    ),
  .rst_n    (rst_n  ),
  .clear    (clear  ),
  .i_data_re (cic_re ),
  .i_data_im (cic_im ),
  .i_valid  (cic_valid),
  .o_data_re (agc_re ),
  .o_data_im (agc_im ),
  .o_mean_abs(mean_abs ),
  .o_valid  (agc_valid)
);
```

```
  reg [9:0] symb_period_cnt;
  wire agc_stable;
  wire sampler_enb;
  always @(posedge clk, negedge rst_n) begin : conter_u
    if(!rst_n || clear) begin
      symb_period_cnt <= 0;
    end else if (enb && lpf_valid) begin
      symb_period_cnt <= symb_period_cnt + 1;
    end
  end
  assign agc_stable = symb_period_cnt >= AGC_DELAY;
  assign sampler_enb = agc_stable && agc_valid;

  sampler #(
    .DIN_W      (AGC_W      ),
    .SMPL_PER_SYMB(4),
    .SMPL_PER_SYMB_W(2),
    .AVG_LENGTH(32),
    .TH_HOLD_HIGH(3*2**10),
    .TH_HOLD_LOW(2**8),
    .DELAY_W(8),
    .BUFFER_LEN(4),
    .BUFFER_LEN_W(2)
  ) sampler_u (
    .clk    (clk       ),
    .i_data (agc_im    ),
    .i_valid(sampler_enb),
    .enb    (enb       ),
    .rst_n  (rst_n     ),
    .clear  (clear     ),
    .o_symbl(symbl     ),
    .o_symb_lock(o_symb_lock),
    .o_valid(symbl_valid)
  );

  assign o_valid    = (agc_stable)? symbl_valid: dec_valid;
  assign o_mean_abs = mean_abs;
  assign o_symbl    = (agc_stable)? symbl: {AGC_W{1'b0}};

//DECIMATION
  cic_filter #(
    .WIDTH_IN (AGC_W),
    .WIDTH_OUT(DEC_W),
    .DECIRATE (4    ),
    .DIFF_DLY (1    ),
    .COUNT_W  (2    )
  ) decimation_u (
    .clk    (clk     ),
    .rst_n  (rst_n   ),
    .enb    (enb     ),
    .clear  (clear   ),
    .i_x_re (agc_re  ),
    .i_x_im (agc_im  ),
    .i_valid(agc_valid),
    .o_valid(dec_valid),
    .o_y_re (dec_re  ),
    .o_y_im (dec_im  )
  );
```

```
assign dec_rs_re = dec_re;
 assign dec_rs_im = dec_im;

cordic_it #(
  .DATA_W(CORDIC_W),
  .NUM_OF_IT(8),
  .II_W(3),
  .GAIN_ADJ(0)
) cordic_it_u (
  .clk          (clk),
  .rst_n        (rst_n),
  .enb          (enb),
  .sink_valid   (dec_valid),
  .vec_mod      (1'b1),
  .i_x          (dec_rs_re),
  .i_y          (dec_rs_im),
  .i_theta      ({CORDIC_W{1'b0}}),
  .source_ready (1'b1),
  .sink_ready   (cordic_ready),
  .o_x          (cordic_x),
  .o_y          (cordic_y),
  .o_theta      (cordic_theta),
  .source_valid (cordic_valid)
);

  loop_filter #(
    .DIN_W (CORDIC_W),
    .DOUT_PREC(FREQ_W),
    .DOUT_W(LPF_W  ),
    .COEF_W(8),
    .KP(68),
    .KI(91),
    .KP_FRAC(16),
    .KI_FRAC(21)
  ) lpf_u (
    .rst_n  (rst_n      ),
    .clk    (clk        ),
    .enb    (enb        ),
    .clear  (clear      ),
    .i_valid(cordic_valid),
    .din    (cordic_theta),
    .o_valid(lpf_valid  ),
    .dout   (lpf        )
  );

 // Frequency desviation register (demod loop start and end point)
 always @(posedge clk, negedge rst_n) begin : dfreq_reg_u
   if(!rst_n || clear) begin
      dfreq <= {LPF_W{1'b0}};
   end else if (enb) begin
      if (lpf_valid) dfreq <= lpf;
   end
 end

 assign o_freq = dfreq + i_freq;

endmodule
```