MINISTÉRIO DA CIÊNCIA E TECNOLOGIA
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

DESENVOLVIMENTO DE ALGORITMOS PARA DECODIFICAÇÃO DO SINAL DO SISTEMA BRASILEIRO DE COLETA DE DADOS

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 2016

MINISTÉRIO DA CIÊNCIA E TECNOLOGIA
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

DESENVOLVIMENTO DE ALGORITMOS PARA DECODIFICAÇÃO DO SINAL DO SISTEMA BRASILEIRO DE COLETA DE DADOS

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 2016

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 para RTL blocos de Processamento Digital de Sinal (PDS) que compõem o modelo em MatLab do decodificador para o sinal do Sistema Brasileiro de Coleta de Dados Ambientais. Este documento apresenta o código dos blocos PDS no nível RTL (*Register Transfer Level*) e o resultado das simulações que foram feitos até o momento. Os blocos desenvolvidos e testados até então foram o filtro CIC e o algoritmo CORDIC. Aos serem testados foram comparados ao modelo MatLab e ambos exibiram compatibilidade com os resultados do modelo.

SUMÁRIO

INTRODUÇÃO

Este trabalho, iniciado em abril de 2016, tem como objeto dar continuidade ao projeto de Iniciação Científica em andamento desde 2013, que consiste em desenvolver um modelo em MatLab de um decodificador para o sinal do Sistema Brasileiro de Coleta de Dados Ambientais (SBCDA). A proposta era de o modelo ser desenvolvido utilizando algoritmos de baixa complexidade computacional para facilitar uma futura implementação do mesmo em um nanossatélite. Para validar o sistema, um ambiente de simulação em MatLab também estava previsto. Este plano de trabalho inicial foi concluído em março de 2016. Assim, o trabalho atual passou a ser iniciar a conversão desse modelo em MatLab para um modelo em RTL, objetivando uma implementação em FPGA. Foram feitos até agora códigos HDL para implementar algumas funções de processamento digital de sinal que são usadas pelo sistema, como o filtro CIC e o algoritmo CORDIC. Estes foram validados a partir de Testbenches, que são códigos para verificar se o design implementado corresponde ao esperado. A estratégia para validação adotada neste trabalho foi comparar os resultados obtidos com os códigos HDL com os obtidos com o modelo em MatLab. As próximas seções irão conter descrições acerca do funcionamento do filtro CIC e do algoritmo CORDIC, bem como seus respectivos códigos em nível RTL (apêndice).

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.

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

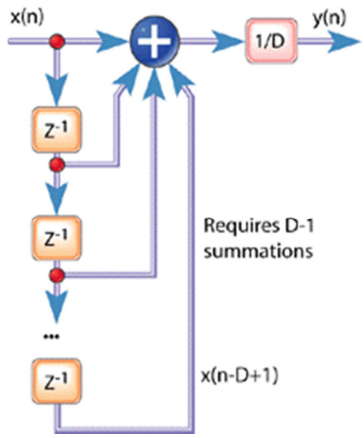

Figura 1 Filtro Média Móvel Não Recursiva

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:

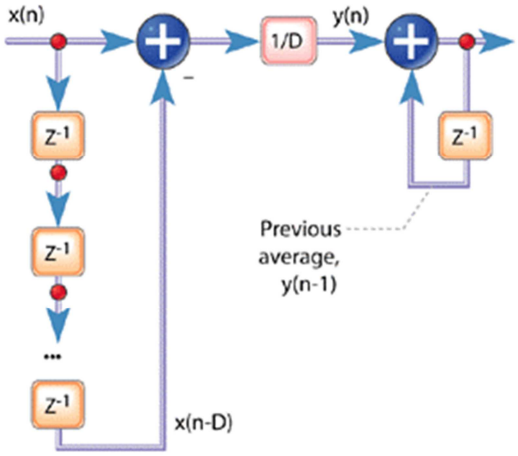

Figura 2 Filtro Média Móvel Recursiva

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:

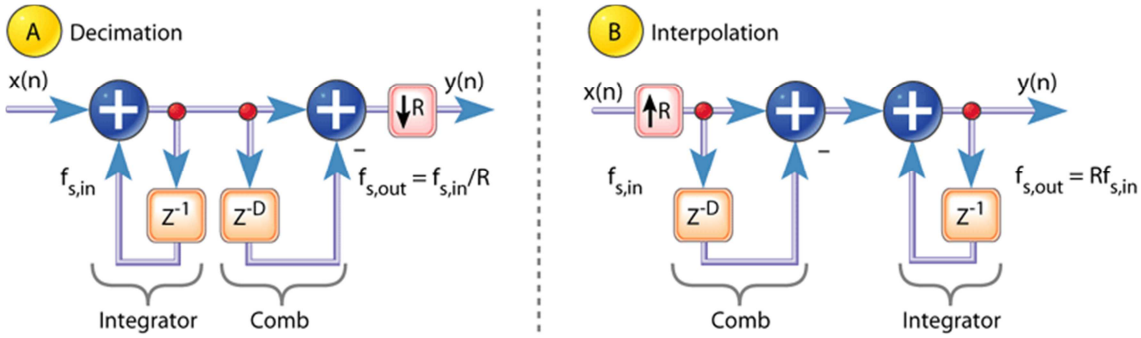

Figura 3 Filtro CIC

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:

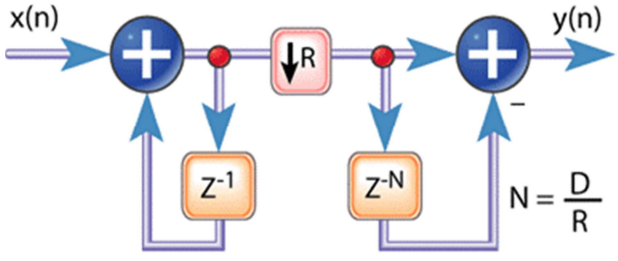

Figura 4 Filtro CIC de primeira ordem - Decimação

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

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 do parâmetro R dependendo da utilização dentro do projeto final.

CORDIC ROTATION/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.

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

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 rotação, o acumulador angular é inicializado com o ângulo de rotação desejado. A decisão de rotação a cada iteração é feita de modo a diminuir a magnitude do ângulo residual no acumulador angular. A decisão a cada iteração é, portanto, baseada no índice do ângulo residual depois de cada passo. As equações para o modo rotação são:

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

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

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

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

$$x_n = A_n[x_0 cos(z_o) - y_0 \sin(z_0)]$$

$$y_n = A_n[y_0 cos(z_o) + x_0 \sin(z_0)]$$

$$z_n = 0$$

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

No modo vetorização, 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:

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

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

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

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 $\pm 2\pi$. 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.

Alternativamente, pode-se fazer uma rotação de 0 ou π, prevenindo a mudança das componentes x e y. Para ambos os casos não há ganho na rotação inicial:

$$x' = d * x$$

$$y' = d * y$$

$z' = z$ se d =1, ou $z - \pi$ se d = -1

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

A primeira redução é mais consistente com rotações sucessivas, enquanto a segunda pode ser mais conveniente quando a fiação é restrita, como em muitos casos com FPGAs.

CONCLUSÃO

Com este relatório é possível compreender o funcionamento do filtro CIC e do CORDIC que fazem parte dos blocos PDS do modelo em MatLab que já foram convertidos para um modelo em RTL. A validação dos códigos convertidos foi feita tomando como base os resultados do modelo em MatLab, comparando-os aos resultados obtidos dos códigos HDL. Os resultados obtidos foram compatíveis. Em anexo encontram-se os códigos HDL dos blocos citados.

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 ed.). Prentice Hall.

APÊNDICE

FILTRO CIC:

```
module cicFilter
#(
  parameter width_in = 16,
  parameter width_out = 32,
  parameter deciRate = 40
)
(
  input wire clk,
  input wire rst_n,
  input wire signed [width_in-1:0] i_x_re,
  input wire signed [width_in-1:0] i_x_im,
  output reg o_valid,
  output reg signed [width_out:0] o_y_re,
  output reg signed [width_out:0] o_y_im
);

// integrator stage register
reg signed [width_out:0] acc_re;
reg signed [width_out:0] acc_im;

//decimation+COMB
reg signed [width_out:0] dly_re;
reg signed [width_out:0] dly_im;
reg [15:0] count;

always @(posedge clk or negedge rst_n) begin
  if (!rst_n) begin
    acc_re <= 0;
    acc_im <= 0;
    count <= 39;
    o_y_re <= 16'b0;
    o_y_im <= 16'b0;
    dly_im <= 0;
    dly_re <= 0;
    o_valid <= 1'b0;
  end
//accumulator
  else begin

    acc_re <= i_x_re + acc_re;
    acc_im <= i_x_im + acc_im;
    count <= count + 16'd1;
    o_valid <= 1'b0;

//decimation + COMB
    if (count == deciRate-1) begin
      count <= 16'b0;
      o_valid <= 1'b1;
      dly_re <= acc_re;
      dly_im <= acc_im;
```

```
      o_y_re <= acc_re - dly_re;
      o_y_im <= acc_im - dly_im;
    end
  end
end
endmodule
```

CORDIC ROTATION:

```
module cordic_rot
#(
   parameter THETA_W  = 16,
   parameter NUM_OF_IT  = 12,
   parameter II_W  = 4
)
(
   input clk,
   input rst_n,
   input enb,
   input sink_valid,
   input signed [THETA_W-1:0] i_x_rot,
   input signed [THETA_W-1:0] i_y_rot,
   input signed [THETA_W-1:0] i_theta_rot,
   input source_ready,
   output sink_ready,
   output signed [THETA_W-1:0] o_x_rot,
   output signed [THETA_W-1:0] o_y_rot,
   output signed [THETA_W-1:0] o_theta_rot,
   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 [THETA_W-1:0] x_rot;
  reg signed [THETA_W-1:0] y_rot;
  reg signed [THETA_W-1:0] theta_rot;

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

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

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

  // from aritimetic unit
  reg signed [THETA_W-1:0] x_n;
  reg signed [THETA_W-1:0] y_n;
```

```
  reg signed [THETA_W-1:0] theta_n;
  reg signed [THETA_W-1:0] d_x;
  reg signed [THETA_W-1:0] d_y;
  wire signed [THETA_W-1:0] rom_data [0:NUM_OF_IT-1];
  wire signed [THETA_W-1:0] rom_out;
  wire signed [THETA_W-1:0] angle_in;

//-------------------------------------------------------------------------//
//-----------------Assignments output----------------------------------//
assign o_x_rot = x_reg;
assign o_y_rot = y_reg;
assign o_theta_rot = theta_reg;
//-------------------------------------------------------------------------//

//-----------------ROM-----------------------------------------------//
assign rom_data[0] = 16'b0010000000000000 ;
assign rom_data[1] = 16'b0001001011100100 ;
assign rom_data[2] = 16'b0000100111111011 ;
assign rom_data[3] = 16'b0000010100010001 ;
assign rom_data[4] = 16'b0000001010001011 ;
assign rom_data[5] = 16'b0000000101000110 ;
assign rom_data[6] = 16'b0000000010100011 ;
assign rom_data[7] = 16'b0000000001010001 ;
assign rom_data[8] = 16'b0000000000101001 ;
assign rom_data[9] = 16'b0000000000010100 ;
assign rom_data[10] = 16'b0000000000001010 ;
assign rom_data[11] = 16'b0000000000000101 ;

assign rom_out = rom_data[ii];

assign angle_in = 16'b1000000000000000; // pi
//-------------------------------------------------------------------------//


always @(*) begin
 //initial rotation (0 or -pi)
      if(i_x_rot[THETA_W-1] == 1) begin
        //-pi
        x_rot = -i_x_rot;
        y_rot = -i_y_rot;
        theta_rot = i_theta_rot - angle_in;
      end else begin
        //0
        x_rot = i_x_rot;
        y_rot = i_y_rot;
        theta_rot = i_theta_rot;
    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 (theta_mux[THETA_W-1] == 0) 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 or 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 or negedge rst_n)
begin: outreg_u
   if (!rst_n) begin
      x_reg <= 0;
      y_reg <= 0;
      theta_reg <= 0;
      source_valid <=1'b0;
   end
   else if(enb) begin
      x_reg <= x_n;
      y_reg <= y_n;
```

```
    theta_reg <= theta_n;
    source_valid <= count_last;
  end
end
// FSM for sink_ready generation

assign sink_ready = state[1];

always @(posedge clk or 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
```

CORDIC VECTORING:

```
module  cordic_vec #(
  parameter DISCRETE_K = 16, // cordic gain ajustment
  parameter DIV_K = 5,       // shift left to compensate overgaing
  parameter INT_NUM = 8,     // number of cordic interation
  parameter XY_PREC = 8,     // xy input and output precision
  parameter THETA_PREC = 8,  //

  parameter PIPELINE = 1     // 0: No intermediate register
                             // 1: 1 intermediate register
                             // Use re-time on syntesis to improve the
                             // intermediate register postion

)
(
  input clk,
  input rst_n,
  input sink_valid,
  input signed [XY_PREC-1:0] x_in,
  input signed [XY_PREC-1:0] y_in,
  output reg source_valid,
  output reg [INT_NUM:0] theta_out,
  output reg signed [XY_PREC:0] x_out,
  output reg signed [XY_PREC:0] y_out
);


//localparam MULT_PREC = 2*XY_PREC+1;
localparam MULT_PREC = 2*(XY_PREC+1);

// x and y wordlength is increased to acommodate CORDIC gain
reg signed [XY_PREC+2-1:0] x0 [0:INT_NUM];
reg signed [XY_PREC+2-1:0] y0 [0:INT_NUM];
reg [INT_NUM:0] theta0;

reg valid1;
reg signed [XY_PREC+2-1:0] x1;
reg signed [XY_PREC+2-1:0] y1;
reg signed [INT_NUM:0] theta1;

wire signed [MULT_PREC-1:0] mult_x;
wire signed [MULT_PREC-1:0] mult_y;

always @(*) begin
  // Initial rotation (0 or -pi rad)
  if (y_in <= 0) begin  // rotate 0
    theta0[INT_NUM] = 1'b1;
    x0[0] = -y_in;
    y0[0] = x_in;
  end else begin    // rotate -pi
    theta0[INT_NUM] = 1'b0;
    x0[0] = y_in;
    y0[0] = -x_in;
  end
```

```
end

generate
genvar g1;
for (g1=0; g1<INT_NUM; g1=g1+1) begin
  always @(*) begin
    if (y0[g1] < 0) begin
      x0[g1+1] = x0[g1] - (y0[g1] >>> g1);
      y0[g1+1] = y0[g1] + (x0[g1] >>> g1);
      theta0[INT_NUM-(g1+1)] = 1'b1;
    end else begin
      x0[g1+1] = x0[g1] + (y0[g1] >>> g1);
      y0[g1+1] = y0[g1] - (x0[g1] >>> g1);
      theta0[INT_NUM-(g1+1)] = 1'b0;
    end
  end
end
endgenerate

generate
if(PIPELINE==0) begin
// NO PIPELINE STAGE
  always @(*) begin
    x1 = x0[INT_NUM];
    y1 = y0[INT_NUM];
    theta1 = theta0;
  end
end else begin
// PIPELINE STAGE
  always @(posedge clk) begin
    x1 <= x0[INT_NUM];
    y1 <= y0[INT_NUM];
    theta1 <= theta0;
  end
end
if(PIPELINE) begin
  always @(posedge clk, negedge rst_n) begin
    if(!rst_n) begin
      valid1 <= 1'b0;
    end else begin
      valid1 <= sink_valid;
    end
  end
end else begin
  always @(*) begin
    valid1 = sink_valid;
  end
end
endgenerate


assign mult_x = (x1 * DISCRETE_K);
assign mult_y = (y1 * DISCRETE_K);

// Outptut register
```

```
always @(posedge clk, negedge rst_n) begin
  if(!rst_n) begin
    source_valid <= 1'b0;
    x_out <= 0;
    y_out <= 0;
    theta_out <= 0;
  end
  else begin
    //Cordic gain compensation
    source_valid <= valid1;
    if(valid1) begin
      x_out <= (mult_x >>> DIV_K);
      y_out <= (mult_y >>> DIV_K);
      theta_out <= theta1;
    end
 end
end
endmodule
```