AGH University of Science and Technology
Cracow
Department of Electronics

Custom system design in FPGA laboratory

Tutorial 1

The Sequential cordic processor for sine &cosine calculations.

Author: Paweł Russek

ver. 2018.03.06

1. Introduction

1.1. Objectives

The main goal of this tutorial is to walk the student through the design steps of the simple sequential custom processor. We will end up with the RTL (ang. Register Transfer Level) description of the custom processor for the sin/cos calculations. The RTL is most robust and commonly used HDL (ang. Hardware Description Language) coding style that is suitable for hardware synthesis i.e. creation of the hardware architecture.

As an example, we will use the vary popular and widely used cordic algorithm for sin/cos calculation, and consequently we will create the hardware description of cordic. The description will be delivered in Verilog HDL.

1.2. Prerequisites

Digital design background will be necessary for this tutorial. We expect the student to know principles of operation of synchronious sequential circuits. Also, the understanding of basic concepts that are foundation of any HDL will be necessary for this tutorial. The prior Verilog knowledge will be helpful but not absolutely necessary.

2. Cordic

Cordic (COordinate Rotation DIgital Computer) is an universal tool for calculation of the trigonometric and hyperbolic functions e.g. sin, cos, arctan, tanh. We will present cordic in rotation mode which is used for sin/cos calculation.

It is an iterative algorithm that replaces the multiplication/division by shift operations and therefore is useful for efficient hardware implementation.

In principle, cordic starts with angle $\vartheta(0) = 0$, and converge to the target angle Θ by adding and subtracting decreasing angle values $\alpha(0) > \alpha(1) > \ldots > \alpha(N-1)$, where $\vartheta(n+1) = \vartheta(n) + \alpha(n)$, and $\vartheta(N) \approx \Theta$. $\sin(\Theta)$ and $\cos(\Theta)$ are co-products of the iteration process. The details of the algorithm will follow.

2.1. Alternate vector representation

For the unit vector in Cartesian coordinates, we can replace vector coefficients $\begin{bmatrix} x \\ y \end{bmatrix}$ by the corresponding sinus and cosinus values of the angle between the vector and the X axis i.e. $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \cos\vartheta \\ \sin\vartheta \end{bmatrix}$.

It is depicted in Figure 1.

Consequently, to know sinus and cosinus values of the angle ϑ it is equivalent to know the coordinates of the unit vector which form an angle ϑ with the X axis.

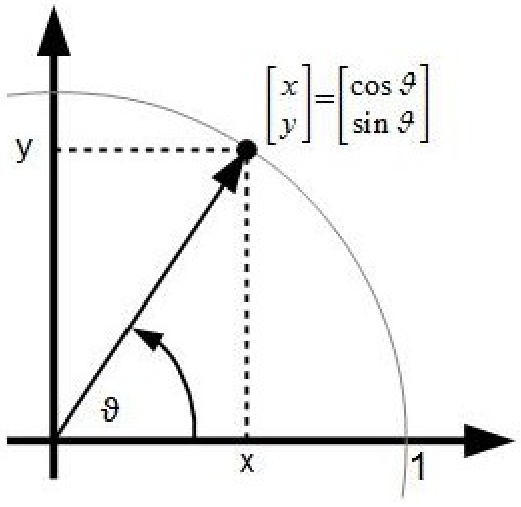

$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \cos\vartheta \\ \sin\vartheta \end{bmatrix}$
y
ϑ
x
1

Figure 1

2.2. Vector rotation

Given the vector $\begin{bmatrix} x0 \\ y0 \end{bmatrix}$, we can perform its rotation by the α angle using the matrix multiplication:

$$\begin{bmatrix} x1 \\ y1 \end{bmatrix} = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} x0 \\ y0 \end{bmatrix}$$

By extracting cos (α), we can also express the rotaton as

$$\begin{bmatrix} x1 \\ y1 \end{bmatrix} = \cos\alpha \begin{bmatrix} 1 & -\tan\alpha \\ \tan\alpha & 1 \end{bmatrix} \begin{bmatrix} x0 \\ y0 \end{bmatrix} .$$

It is also worth to note that the vector rotation by angle (-α) can be express by means of angle α as

$$\begin{bmatrix} x1 \\ y1 \end{bmatrix} = \cos\alpha \begin{bmatrix} 1 & \tan\alpha \\ -\tan\alpha & 1 \end{bmatrix} \begin{bmatrix} x0 \\ y0 \end{bmatrix}$$

because cos(- α) = cos (α), and tan(- α) = - tan(α).

2.3. Successive vector approximation

If we start with vector $v(0) = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, we can approach any vector $V = \begin{bmatrix} \cos\vartheta \\ \sin\vartheta \end{bmatrix}$ thanks to the successive left or right rotations of the vector v by the predefined values of angles α(0) > α(1) > α(2) > ... > α(N-1).

$$v(1) = rotate(v(0))$$
$$\dots$$
$$v(n) = rotate(v(n-1))$$
$$\dots$$
$$v(N) = rotate(v(N-1)) \approx V$$

At each iteration step, the vector angle is compared with the target angle ϑ. If the angle is smaller then ϑ, the next rotation is in the left direction. If the angle is greather then ϑ, the next rotation is in the right direction. Figure 2 presents the example iteration process.

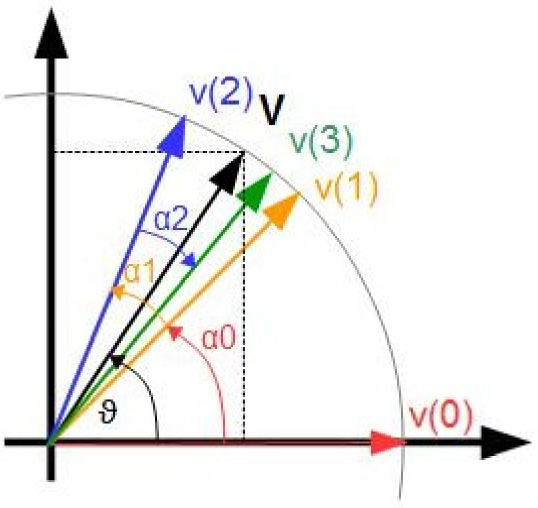


Figure 2

We can write the following formula for the v(n) vector coordinates.

$$\begin{bmatrix} x_n \\ y_n \end{bmatrix} = \cos(\alpha_{n-1}) \begin{bmatrix} 1 & -\tan\alpha_{n-1} \\ \tan\alpha_{n-1} & 1 \end{bmatrix} \begin{bmatrix} x_{n-1} \\ y_{n-1} \end{bmatrix}$$

The angle α is positive for left rotation and negative for right rotation. Tangent α changes its sign depends on the rotation direction, but, importantly for cordic, cosinus α is independent of the rotation direction.

2.4. Cordic trick

What makes cordic attractive for hardware implementation is that we can get rid of multiplication in the computation of the next vector position.

If the angles α(0) > α(1) > α(2) > ... > α(N-1) are selected in a way that $\tan\alpha_n = \pm\frac{1}{2^n}$ (plus for left, and minus for right rotation), we can replace division by the shift right operation in matrix multiplication. The aggregated multiplication by $\cos\alpha_n$ form a scaling factor K_N which is constant for the given number of iteration N.

$$K_N = \prod_{n/0}^{N-1} \cos\alpha_n \quad .$$

Thus, we have the formula to calculate sinus and cosinus of ϑ:

$$\begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix} = K_N \begin{bmatrix} 1 & \mp\tan\alpha_{N-1} \\ \pm\tan\alpha_{N-1} & 1 \end{bmatrix} \cdots \begin{bmatrix} 1 & \mp\tan\alpha_n \\ \pm\tan\alpha_n & 1 \end{bmatrix} \cdots \begin{bmatrix} 1 & -\tan\alpha_0 \\ \tan\alpha_0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad ,$$

where $\tan\alpha_n = \frac{1}{2^n}$.

Plus and minus sign before *tan* is derived in iteration process according to comparison of the current running angle to the target angle ϑ.

2.5. Algorithm

The program for cordic algorithm in the Verilog-like code syntax is given below. Note that for the different number of iterations, the *Kn* scaling factor must be adjusted,

```
module cordic_beh();
/**
* Cordic algorithm
*/
real t_angle = 1.0; //Input parameter. The angle

//Table of arctan (1/2^i)
// Note. Table initialization below is not correct for Verilog. Select System-Verilog mode
// in your simulator in the case of syntax errors
real arctan[0:10] = { 0.785398163, 0.463647609, 0.244978663, 0.124354995, 0.06241881,
                      0.031239833, 0.015623729, 0.007812341, 0.00390623,  0.001953123,
                      0.000976562 };
real Kn = 0.607253; //Cordic scaling factor for 10 iterations

//Variables
real cos = 1.0; //Initial vector x coordinate
real sin = 0.0; //and y coordinate
real angle = 0.0; //A running angle

integer i, d;
real tmp;

initial //Execute only once
begin
    for ( i = 0; i < 11; i = i + 1) //Ten algorithm iterations
    begin
        if( t_angle > angle )
        begin
            angle = angle + arctan[i];
            tmp = cos - ( sin / 2**i );
            sin = ( cos / 2**i ) + sin;
            cos = tmp;
        end
        else
        begin
            angle = angle - arctan[i];
            tmp = cos + ( sin / 2**i );
            sin = - ( cos / 2**i) + sin;
            cos = tmp;
        end //if
    end //for
    //Scale sin/cos values
    sin = Kn * sin;
    cos = Kn * cos;
    $display("sin=%f, cos=%f", sin, cos);
end
endmodule;
```

Exercise 2.1

Run the module cordic_beh in the Verilog simulator. Change the number of iterations to improve algorithm accuracy. Compare the calculated sin/cos results to the real values.

3. Fixed-point data representation

An efficient way to represent real numbers in custom processor architecture is fixed-point representation. Fixed-point can be seen as a scaled real number that is rounded to integer. We prefer scaling factors that are powers of two in binary systems. For example for the scaling factor 256 we represent 3.14 as $\lfloor 3.14\times 256\rfloor = 803 = 0x323$.

The fixed point format is specified as [m|n] , where m is the width of the number representation, and n is **binary point** position within the number. For example [12|8] fixed-point representation of 3.14 is:

bit	**b11**	**b10**	**b9**	**b8**	**b7**	**b6**	**b5**	**b4**	**b3**	**b2**	**b1**	**b0**
weigth	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
3.14	0	0	1	1	0	0	1	0	0	0	1	1
hex	3				2				3			

3.1. Fixed-point addition

In fixed-point format addition is performed the same way as for natural binary code. The sum of two fixed [m|n] numbers is also a fixed [m+1|n] number (plus one bit for carry out bit).

For example:

$$3.14+2.71 \approx (803)_{[12|8]}+(693)_{[12|8]}=(1496)_{[12|8]}=1496/256=5.84375$$

3.2. Fixed-point multiplication

Multiplying the two fixed-point numbers results in a binary point position shift. When two fixed [m|n] numbers are multiplied, the result format is the [2*m|2*n] fixed-point format. Thus, to keep the binary point of the result and operands consistent we might want to shift the results n bit positions right.

For example:

$$3.14\times 2.71\approx(803)_{[12|8]}\times(693)_{[12|8]}=(556479)_{[24|16]}=(0x87DBF)_{[24|16]}=(0x87D)_{[24|8]}=(2173)_{[24|8]}=$$
$$2173/256=8.4882$$

3.3. Fixed-point division by 2^n

To divide fixed-point by the power of two, we perform the shift right operation. The programmer must be aware to perform arithmetic shift right (msb is preserved) for the signed fixed-point numbers, and the logical shift right (msb is set to zero) for unsigned numbers.

The Verilog operator for arithmetic shift right is >>> operator, in contrast to logical shift right which is >> operator.

For example:

$$-3.14/4 \approx (0b110011011101)_{[12|8]}/4=(0b110011011101)_{[12|8]}>>>2=$$
$$(0b111100110111)_{[12|8]}=-201/256=-0.7851$$

Exercise 2.1

Write the cordic algorithm using the signed fixed-point [12:10] data representation. Use arithmetic shift right for the division operation. Simulate the resulting code in Verilog simulator.

Hint:

1. Use “reg signed [11:0]” data type to declare the fixed-point variables. For example:

```
module cordic_beh_fixedpoint();
parameter integer FXP_SCALE = 1024;

reg signed [11:0] t_angle = 0.8 * FXP_SCALE; //Input angle

reg signed [11:0] cos = 1.0 * FXP_SCALE; //Initial condition
reg signed [11:0] sin = 0.0;
reg signed [11:0] angle = 0.0; //Running angle

reg signed [11:0] atan[0:10] = { .... }

reg signed [11:0] Kn = 0.607253 * FXP_SCALE;

etc ...

endmodule
```

2. Make sure that the result of multiplication: “`Kn * sin;`” “`Kn * cos;`” fits into the output register. It's width must be declared **twice the width** of `Kn, sin, and cos` registers.

4. Cordic processor RTL description

So far, we have examined a **behavioural description** of the cordic algorithm, which is only a formal specification of the desired system output for the given input. No digital architecture of cordic processor was proposed, thus, at the moment, our system is defined as a black box with no internal structure specified..

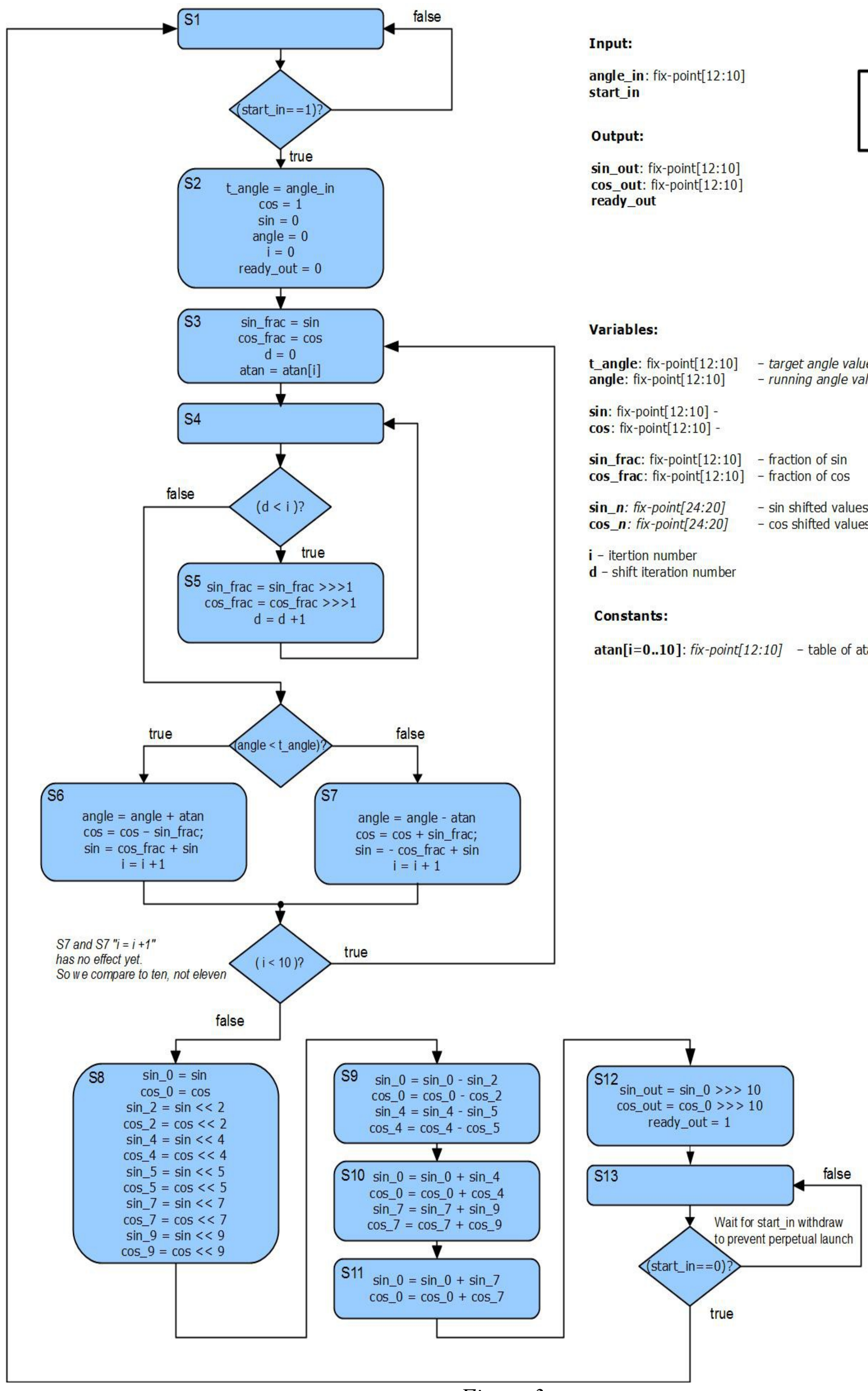
Today, tools for behavioural hardware synthesis exist, however, the goal of main tutorial is to provide RTL (Register Transfer Level) description of the cordic processor. RTL design is a standard for in ASIC and FPGA design today, as it deliver the best results in terms of hardware performance and resource requirements.

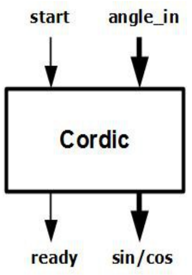
In RTL, we define the digital system architecture by means of data transfer between registers. While data is transferred, well-defined operations are performed. The work is synchronised by a clock signal, therefore, the system transaction are well defined at the given time scale.

4.1. Finite State Machine with Data

To create RTL model of the processor for a given algorithm, FSMD (Finite State Machine with Data) is a good starting point. Basically, FSMD is an automata (FSM) that has data transactions associated with each state. Data is represented by variables and transactions can be e.g. arithmetical or logical operations.

Figure 3 presents an FSMD for cordic.





Input:

angle_in: fix-point[12:10]
start_in

Output:

sin_out: fix-point[12:10]
cos_out: fix-point[12:10]
ready_out

Variables:

t_angle: fix-point[12:10]	– *target angle value*
angle: fix-point[12:10]	– *running angle value*
sin: fix-point[12:10] -	
cos: fix-point[12:10] -	
sin_frac: fix-point[12:10]	– fraction of sin
cos_frac: fix-point[12:10]	– fraction of cos
sin_*n*: *fix-point[24:20]*	– sin shifted values
cos_*n*: *fix-point[24:20]*	– cos shifted values

i – itertion number
d – shift iteration number

Constants:

atan[i=0..10]: *fix-point[12:10]* – table of atan(2**-i)

Figure 3

Basically, the cordic FSMD is self-explanatory but a couple of things may require comments.

The cordic calculations starts in the S2 state, as S1 is necessary to synchronise the cordic processor with the user external circuit. For the synchronisation *start* and *ready* signals are used. The processing starts when the user issues the input *angle* value and signal *start* =1. Processing stops at S12, when the processor sends the calculated sin/cos values to its outputs and issues the *ready* signal. In S13, we wait for *start* withdraw, as we do not want the processor to accidentally start processing again for the same input angle.

The S3, S4, and S5 states are used to calculate fractions of *sin* and *cos* intermediate values i.e. *sin* $* 2^{-i}$, *cos* $* 2^{-i}$. We perform one bit arithmetic shift left in the loop, as we do not want to involve only one-bit-shifter in the cordic processor architecture.

The processor use the fixed-point [12|10] as main data representation. This representation was found to be suitable for the accuracy of the ten-round cordic algorithm.

Fixed-point [24|20] is also use to keep intermediate values of the multiplication by the scaling factor *Kn* (*sin_shift_*n and *cos_shift_*n variables). The K_N=0.607253 multiplication is performed in the S8-S11 states. The binary data representation of K_N=0.607253 is $\lfloor 0.6073 \times 1024 \rfloor = 621 = 0\text{x}26\text{D} = 0\text{b}0010\,0110\,1101$. The processor uses the Booth method to perform multiplication.

4.2. Booth method

The classical method of multiplication *M * C,* where $C = c_N \ldots c_1 c_0$; $c_i = \{0, 1\}$, is a constant value, can be expressed by the formula:

$$M \times C = \sum_{i/0}^{N} c_i M \ll i$$

For example:

$$M = 0\text{b}001101110$$
$$C = 0\text{b}001001001$$
$$c_7 = 1, c_3 = 1, c_0 = 1$$
$$M \times C = 0\text{b}001101110 \ll 7 + 0\text{b}001101110 \ll 3 + 0\text{b}001101110 \ll 0;$$

The Booth method takes advantages of the observation that a chunk of ones in binary representation can be expressed as a simple subtraction:

$$\sum_{i/b}^{e} 2^i = 2^{e+1} - 2^b \quad .$$

For example:

$$0\text{b}0111\ 1100\ 0001\ 1100 = \sum_{i/10}^{14} 2^i + \sum_{i/3}^{5} 2^i = 2^{14} + 2^{13} + 2^{12} + 2^{11} + 2^{10} + 2^5 + 2^4 + 2^3 = 2^{15} - 2^{10} + 2^5 - 2^3 \quad .$$

Note, that seven additions were replaced by two subtraction and one addition in the example.

Thus, in *M*C* multiplication, we can replace consecutive summation by one subtraction operation.

For example:

$$M \times 0b0111\ 1100\ 0001\ 1100 =$$
$$(M \ll 14) + (M \ll 13) + (M \ll 12) + (M \ll 11) + (M \ll 10) + (M \ll 5) + (M \ll 4) + (M \ll 3) =$$
$$2^{15} - 2^{10} + 2^5 - 2^3 = (M \ll 15) - (M \ll 10) + (M \ll 5) - (M \ll 3)$$

In the ten-round cordic algorithm the $M \times K_N = M \times 0b0010\,0110\,1101$ multiplication can be expressed as $M \times K_N = (M \ll 9) + (M \ll 7) - (M \ll 5) + (M \ll 4) - (M \ll 2) + (M \ll 0)$.

Obviously, the presented FSMD is not he only solution for the given cordic behavioural description. One can derive its own approach, which can be faster/slower and more/less logic resource hungry.

Exercise 4.1

Complete and simulate the RTL description of the cordic algorithm that is given below. Use the testbench “cordic_rtl_TB” code that is provided for your convenience.

```
module cordic_rtl( clock, reset, start, angle_in, ready_out, sin_out, cos_out);
parameter integer W = 12; //Fixed-point representation precision fixpoint(2:10)
parameter FXP_MUL = 1024;
parameter FXP_SHIFT = 10;
input clock, reset;
input start; //start processing
input [W-1:0] angle_in;
output reg ready_out; //result is ready
output reg [W-1:0] sin_out, cos_out;

//Cordic look-up table
reg signed [11:0] atan[0:10] = {  12'b001100100100, 12'b000111011011, 12'b000011111011,
                                  12'b000001111111, 12'b000001000000, 12'b000000100000,
                                  12'b000000010000, 12'b000000001000, 12'b000000000100,
                                  12'b000000000010, 12'b000000000001 };
//FSMD states
parameter S1 = 4'h01, S2 = 4'h02, S3 = 4'h03, S4 = 4'h04, S5 = 4'h05,
          S6 = 4'h06, S7 = 4'h07, S8 = 4'h08, S9 = 4'h09, S10 = 4'h0a,
          S11 = 4'h0b, S12 = 4'h0c, S13 = 4'h0d;
reg [3:0] state;
//Algorithm Variables
reg signed [11:0] angle, t_angle, sin, cos, sin_frac, cos_frac;
reg signed [11:0] atan_val;
reg signed [22:0] sin_0, cos_0, sin_2, cos_2, sin_4, cos_4, sin_5, cos_5, sin_7, cos_7,
sin_9, cos_9;
//Iterators
reg [3:0] i, d;

always @ (posedge clock)
begin
    if(reset==1'b1)
    begin
        ready_out <= 1'b0;
        state <= S1;
    end
    else
    begin
        case(state)
            S1: begin
                    if(start == 1'b1) state <= S2; else state <= S1;
                end
            S2: begin
                    t_angle <= angle_in;
                    cos <= 1 * FXP_MUL;
                    sin <= 0;
                    angle <= 0;
                    i <= 0;
                    ready_out <= 0;
                    state <= S3;
                end
            S3: begin
                    sin_frac <= ???;
                    cos_frac <= ???;
                    d <= 0;
                    atan_val <= atan[i];
                    state <= ???;
                end
            S4:begin
                    if( d < i )
                        state <= ???;
                    else
                        if(angle < t_angle) state <= ???; else state <= ???;
                end
            S5:begin
                    sin_frac <= sin_frac >>> 1;
                    cos_frac <= cos_frac >>> 1;
                    d <= ???;
                    state <= ???;
                end
            S6:begin
                    angle <= angle + atan_val;
                    cos <= cos - sin_frac;
                    sin <= cos_frac + sin;
                    i = i +1;
                    if(i < 10) state <= S3; else state <= S8;
                end
            S7:begin
                    angle <= ???;
                    cos <= ???;
                    sin <= ???;
                    i <= i + 1;
                    if(i < 10) state <= ???; else state <= ???;
                end
            S8: begin
                    sin_0 <= sin;
                    cos_0 <= cos;
                    sin_2 <= sin << 2;
                    cos_2 <= cos << 2;
                    sin_4 <= sin << 4;
                    cos_4 <= cos << 4;
                    sin_5 <= sin << 5;
                    cos_5 <= cos << 5;
                    sin_7 <= sin << 7;
                    cos_7 <= cos << 7;
                    sin_9 <= sin << 9;
                    cos_9 <= cos << 9;
                    state <= S9;
                end
            S9: begin
                    sin_0 <= sin_0 - sin_2;
                    cos_0 <= cos_0 - cos_2;
                    sin_4 <= sin_4 - sin_5;
                    cos_4 <= cos_4 - cos_5;
                    state <= S10;
                end
            S10: begin
                    sin_0 <= sin_0 + sin_4;
                    cos_0 <= cos_0 + cos_4;
                    sin_7 <= sin_7 + sin_9;
                    cos_7 <= cos_7 + cos_9;
                    state <= S11;
                end
            S11: begin
                    sin_0 <= sin_0 + sin_7;
                    cos_0 <= cos_0 + cos_7;
                    state <= S12;
                end
            S12: begin
                    sin_out <= sin_0 >>> FXP_SHIFT;
                    cos_out <= cos_0 >>> FXP_SHIFT;
                    ready_out = 1;
                    state <= S13;
                end
            S13: begin
                    if(start == 1'b0) state <= ???; else state <= ???;
                end
        endcase
    end
end
endmodule
```

Testbench for the cordic_rtl module.

```
module cordic_rtl_TB;
reg clock, reset, start;
reg [11:0] angle_in;
wire ready_out;
wire [11:0] sin_out, cos_out;

real real_cos, real_sin;

cordic_rtl cordic( clock, reset, start, angle_in, ready_out, sin_out, cos_out);

//Clock generator
initial
    clock <= 1'b1;
always
    #5 clock <= ~clock;

//Reset signal
initial
begin
        reset <= 1'b1;
    #10 reset <= 1'b0;
end

//Stimuli signals
initial
begin
        angle_in <= 1.5 * 1024;  //Modify value in fixed-point [2:10]
        start <= 1'b0;
    #20 start <= 1'b1;
    #30 start <= 1'b0;
end
```

```
//Catch output
always @ (posedge ready_out)
begin
    #10 real_cos = cos_out;
        real_sin = sin_out;
        real_cos = real_cos / 1024;
        real_sin = real_sin / 1024;
        $display("Real values: sin=%f, cos=%f", real_sin, real_cos);
end

endmodule
```