

VSCPU Project

Report Submitted by

Alok Ranjan (22M1136)



**Department of Electrical Engineering
IIT Bombay**

18 th August 2023

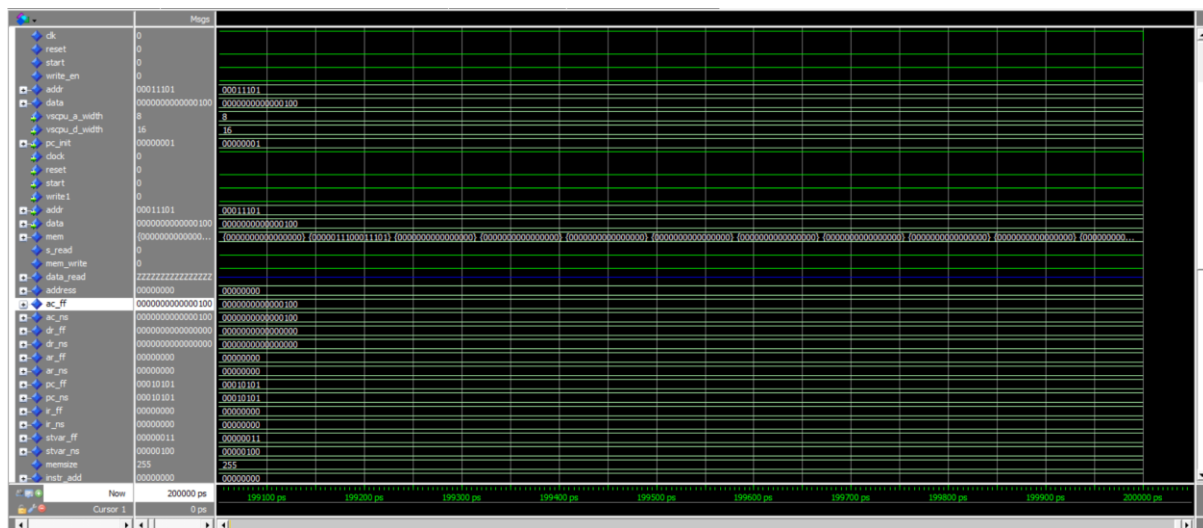
We have developed a processor named VSCPU in VHDL where we defined Width of the Address and Data lines. The memory size was of 255 locations and each location has a size of the width of our data. Our program counter starts at "00000001". The inputs ports of our processor was clock, reset, start, write_en, address and data line. The one output port was the "status".

The four instructions that were applied were - Add, logical AND, Jump and Increment.

1. Load:

- Through the "Load" operation, the content of the memory location with address "00011101" will be stored in the Accumulator
i.e. $AC \leq M[00011101]$
- Data stored in the aforementioned location is "0000000000000100"
i.e. $M[00011101] = 0000000000000100$
- Therefore, after the "Load" operation, the content of the Accumulator is updated to:
 $AC = 0000000000000100$

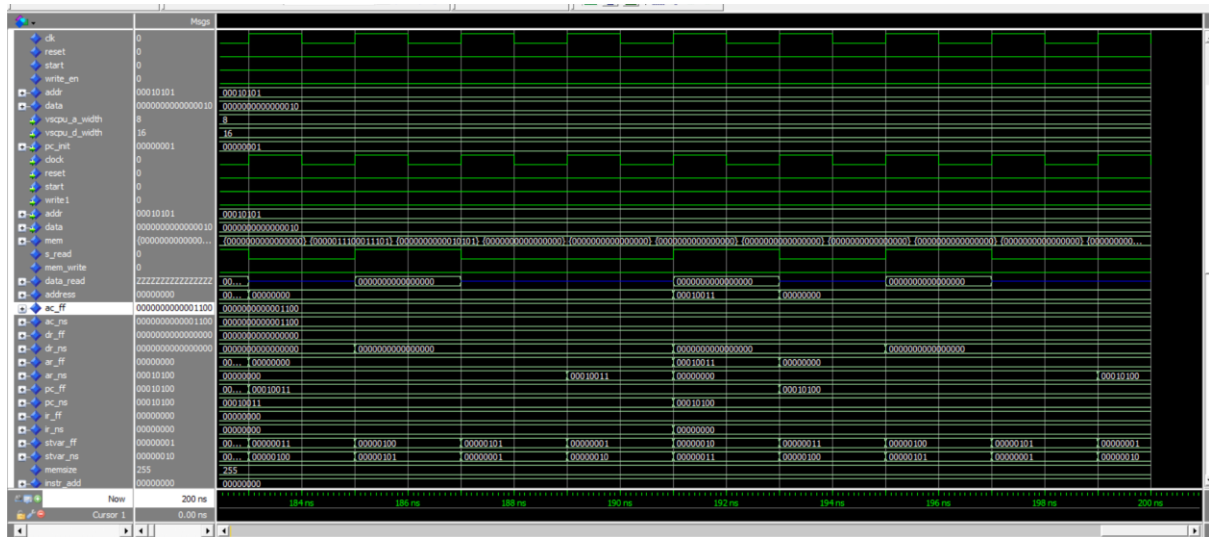
Relevant screenshot attached:



2. Addition:

- $M[00011101] = 00000000000001010$ {i.e. 10 in Decimal}
 $M[00010101] = 0000000000000010$ {i.e. 2 in Decimal}
- $AC \leq M[00011101]$
- $AC = 00000000000001010$ {i.e. 10 in Decimal}
- After addition
 $AC = 00000000000001100$ {i.e. 12 in Decimal}

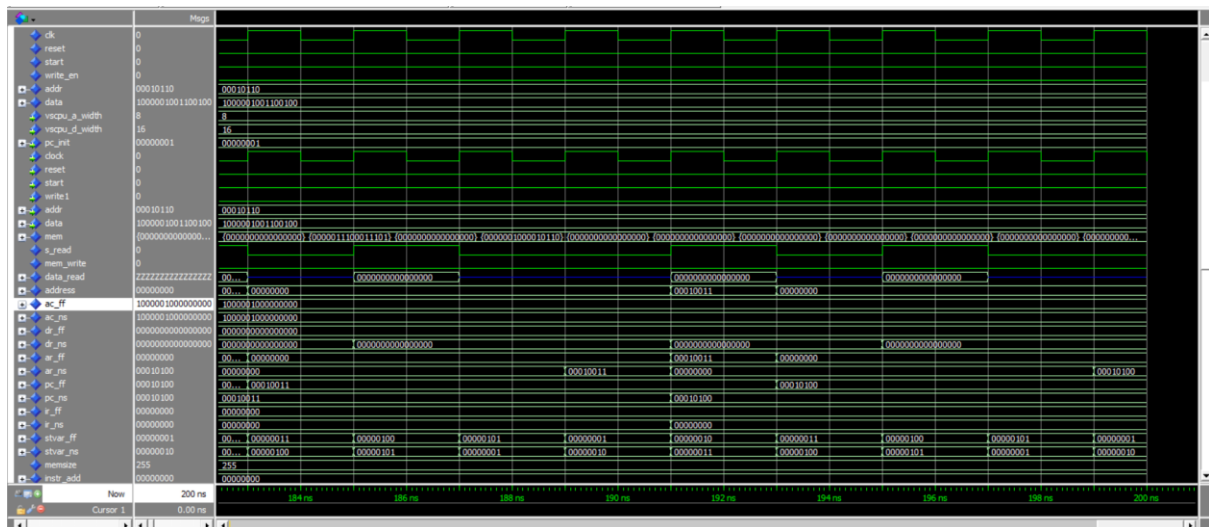
Relevant screenshot attached:



3. AND Operation:

- $M[00011101] = 1000001000001011$
 $M[00010110] = 1000001001100100$
- $AC \leq M[00011101]$
- $AC = 1000001000001011$
- After AND operation
 $AC = 1000001000000000$

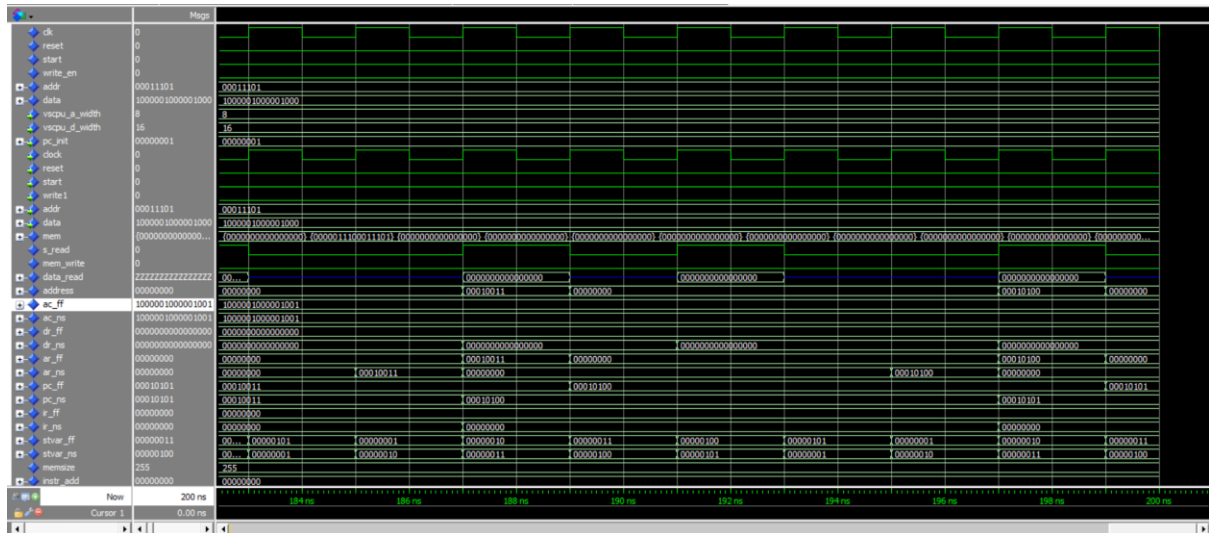
Relevant screenshot attached:



4. Increment:

- $M[00010100] = 1000001000001000$ {i.e. 33288 in Decimal}
- $AC \leq M[00010100]$
- $AC = 1000001000001000$ {i.e. 33288 in Decimal}
- After decrementing
 $AC = 1000001000001001$ {i.e. 33289 in Decimal}

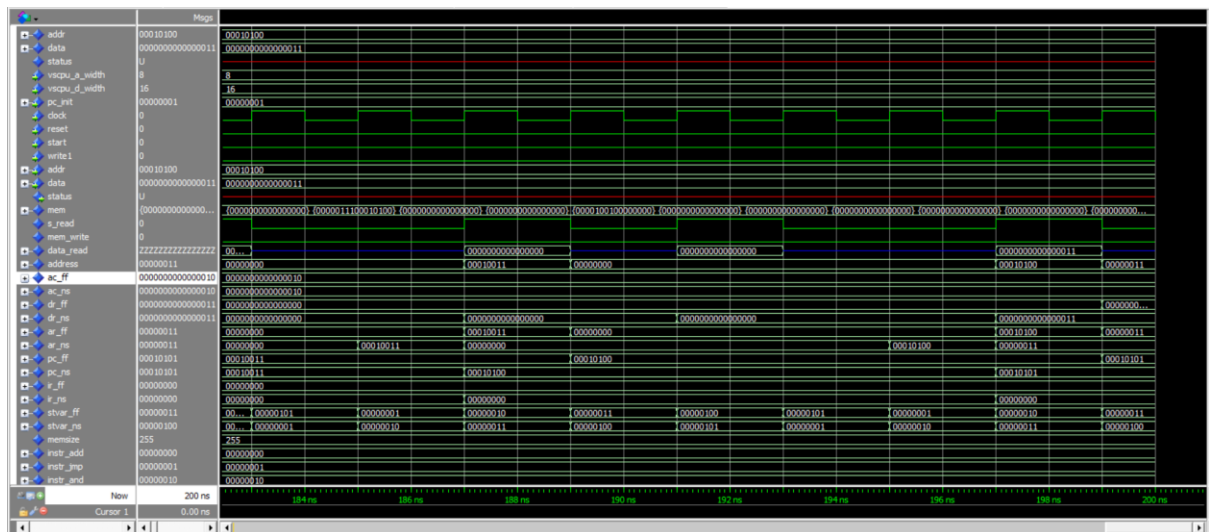
Relevant screenshot attached:



5. Decrement:

- $M[00010100] = 0000000000000011$ {i.e. 3 in Decimal}
- $AC \leq M[00010100]$
- $AC = 0000000000000011$ {i.e. 3 in Decimal}
- After decrementing
 $AC = 0000000000000010$ {i.e. 2 in Decimal}

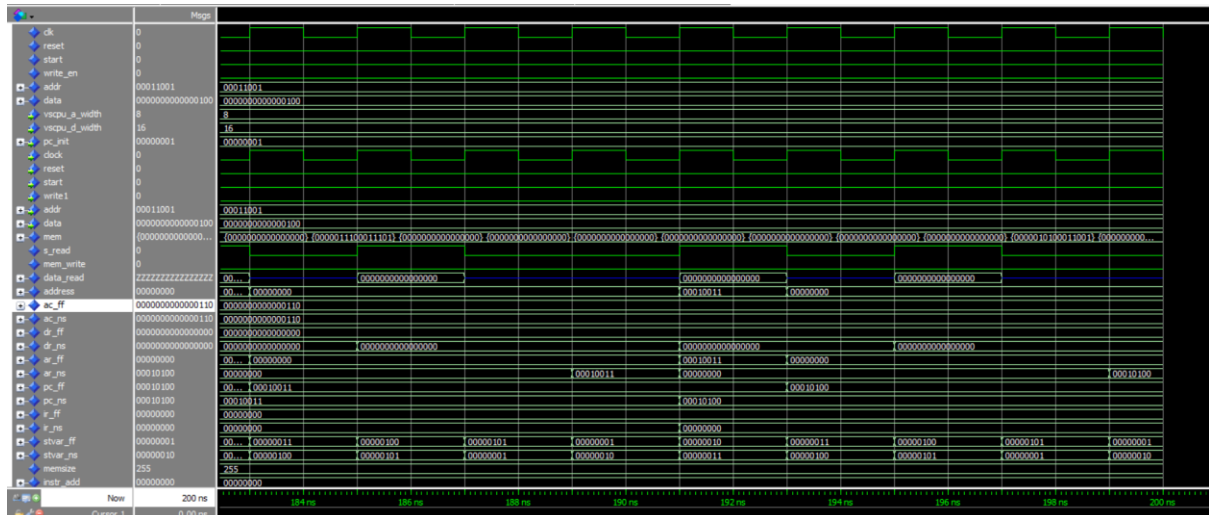
Relevant screenshot attached:



6. Subtraction:

- $M[00011101] = 0000000000001010$ {i.e. 10 in Decimal}
- $M[00011001] = 0000000000000100$ {i.e. 4 in Decimal}
- $AC \leq M[00011101]$
- $AC = 0000000000001010$ {i.e. 10 in Decimal}
- After subtraction
 $AC = 0000000000000110$ {i.e. 6 in Decimal}

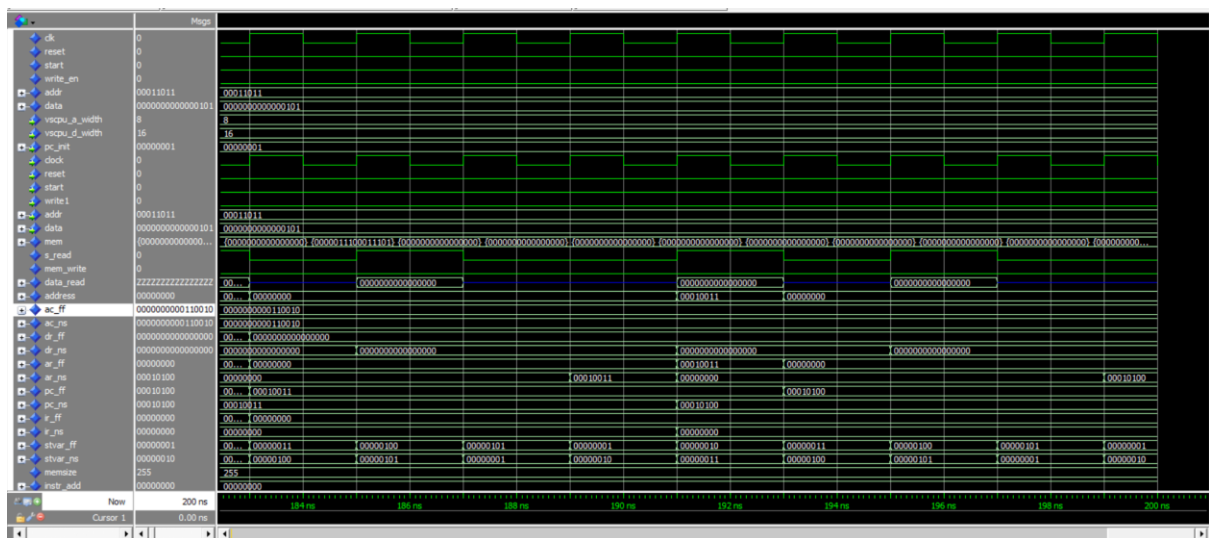
Relevant screenshot attached:



7. Multiplication:

- $M[00011101] = 0000000000001010$ {i.e. 10 in Decimal}
- $M[00011101] = 0000000000000101$ {i.e. 5 in Decimal}
- $AC \leq M[00011101]$
- $AC = 0000000000001010$ {i.e. 10 in Decimal}
- After multiplication
 $AC = 0000000000110010$ {i.e. 50 in Decimal}

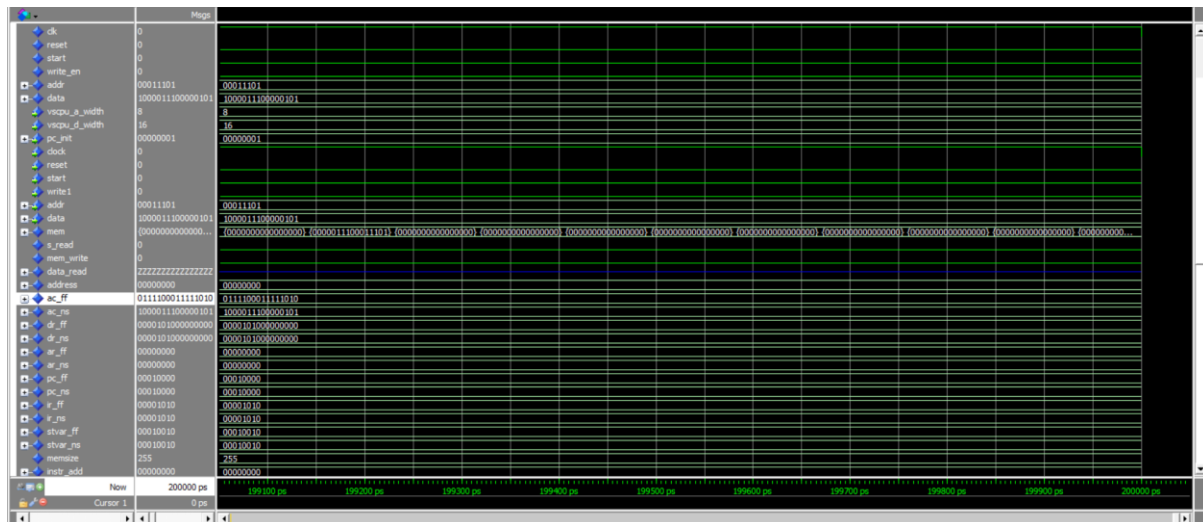
Relevant screenshot attached:



8. Complement:

- $M[00011101] = 1000011100000101$
- $AC \leq M[00011101]$
- $AC = 1000011100000101$
- After Complement operation
 $AC = 0111100011111010$

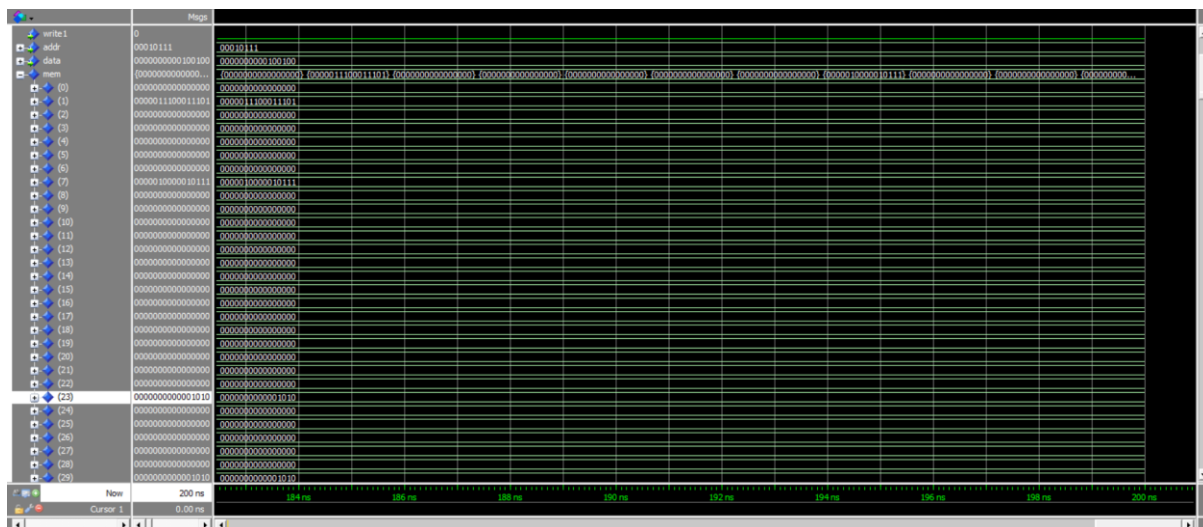
Relevant screenshot attached:



9. Store:

- $M[00011101] = 0000000000001010$ {i.e. 10 in Decimal}
- $M[00010111] = 00000000000100100$
- $AC \leq M[00011101]$
- $AC = 0000000000001010$ {i.e. 10 in Decimal}
- Storing the value of Accumulator in the Memory location 00010111 (23 in Decimal)
- $M[00010111] = 0000000000001010$ {i.e. 10 in Decimal}

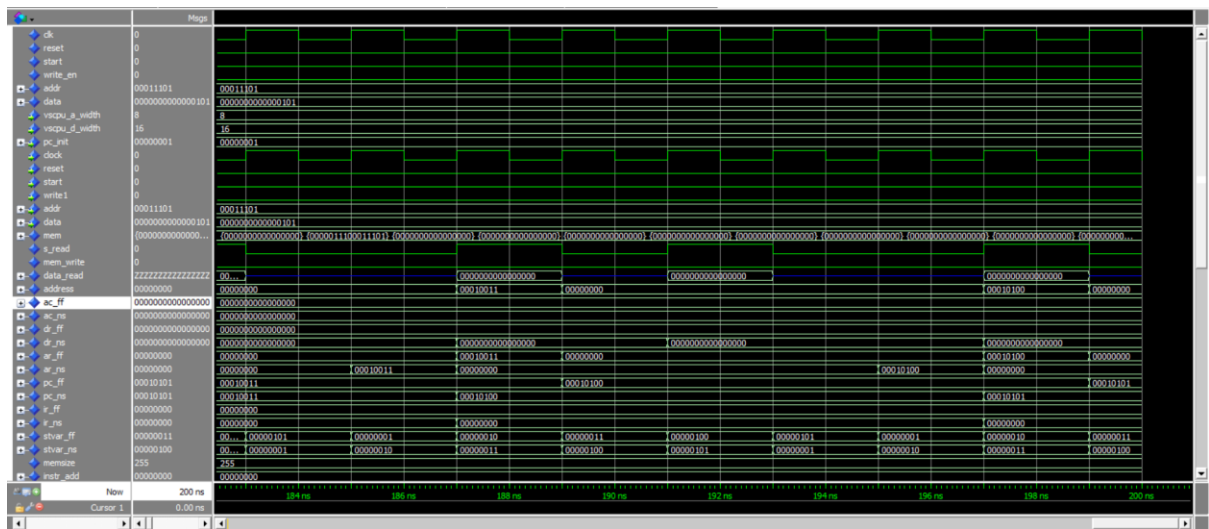
Relevant screenshot attached:



10. Clear operation:

- $M[00011101] = 0000000000000101$
- $AC \leq M[00011101]$
- $AC = 0000000000000101$
- Content of the Accumulator is cleared after the "Clear" operation i.e. $AC = 0000000000000000$

Relevant screenshot attached:



CORDIC

COordinate Rotation DIgital Computer is designed to calculate trigonometric functions using simple operations. The algorithm can also compute division, square root after making small changes. The sine and cosine values for an angle are computed with the basic version. Steps to encode the angle Z0 whose cosine/sine is to be computed:

- The acceptable range of Z0 is $(-90^\circ, 90^\circ)$ and is represented using 16 bits.
- The angle Z0 is converted into radians and the range becomes $(-1.57c, +1.57c)$.
- MSB is '0' for positive angles and '1' otherwise.
- The second most significant bit is the integer part of Z0 in radians. It can be noted that only 1-bit is enough to store the integral part of the angle, provided the acceptable range.
- The fractional part is converted into binary and is stored in 14 bits.

For example,

$Z_0 = +22.5^\circ = +0.3926991 \text{ rad} = 0.01100100100001111111$ (MSB is '0' for positive angle)

$= 0001100100100001111111$ is the input to be given

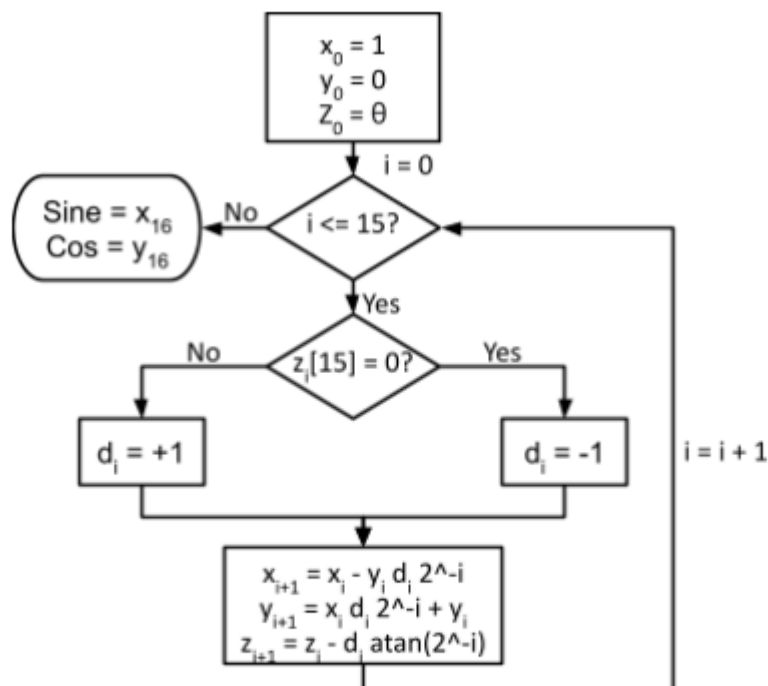


Figure 11: Flowchart of the CORDIC algorithm

Verilog code:

```
module Cordic(clk,rst,x,y,z,S, C);
input clk,rst;
input x,y;
// x=1, y=0 for required mode
input [15:0] z;      // Sign || 0 or 1 || fraction
output [15:0] S,C;

wire [15:0] atan[0:15];
assign atan[0] = 16'b0011001001000100; //stored tan-1 values
assign atan[1] = 16'b0001110110101100;
assign atan[2] = 16'b0000111110101110;
assign atan[3] = 16'b0000011111110101;
assign atan[4] = 16'b0000001111111111;
assign atan[5] = 16'b0000001000000000;
assign atan[6] = 16'b0000000100000000;
assign atan[7] = 16'b0000000010000000;
assign atan[8] = 16'b0000000001000000;
assign atan[9] = 16'b0000000000100000;
assign atan[10] = 16'b0000000000010000;
assign atan[11] = 16'b0000000000001000;
assign atan[12] = 16'b0000000000000100;
assign atan[13] = 16'b0000000000000010;
assign atan[14] = 16'b0000000000000001;
assign atan[15] = 16'b0000000000000000;
reg [15:0] x_r,y_r,z_r;
reg [0:4] count;
reg [15:0] S,C;

always@(posedge clk) begin
    if(rst == 1'b1) begin
        x_r <= {1'd0,x,14'd0};
        y_r <= {1'd0,y,14'd0};
        z_r <= z;
        count <= 5'd0;
    end
    else begin
        if( count <= 15 ) begin
            if (z_r[15] == 0) begin
                x_r <= x_r - (y_r >> count);
                y_r <= y_r + (x_r >> count);
            end
        end
    end
end
```

```

        z_r <= z_r - atan[count];
    end
    else begin
        x_r <= x_r + (y_r >> count);
        y_r <= y_r - (x_r >> count);
        z_r <= z_r + atan[count];
    end
    count <= count+1;
end
else begin
    S <= x_r; // +/- scaled Sine
    C <= y_r; // +/- scaled Cos
end
end
end
endmodule

```

The final outputs x16 and y16 provide us the almost accurate version of cosine and sine values. The obtained outputs are 1.646 times the actual values.

$$x_n = A_n (x_0 \cos(z_0) + y_0 \sin(z_0))$$

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

$$z_n = 0$$

Let us suppose that $Z_0 = 56.5^\circ = 0.986111$ rad, after 16 iterations,

x16 = 0011101000101100 and y16 = 0101011111100101 are obtained.

Steps to decode the outputs:

- The 16-bit outputs also share the same format as the input angle. The MSB specifies the sign.
- The second most significant bit is directly the integral part. The output range is between -1.646 and +1.646 (i.e. $1.646 \times \text{Range}(\cos(Z_0) \text{ or } \sin(Z_0))$).
- The 14 LSBs are to be converted into the fractional part.

For example, obtained cosine value is

$$x_{16} = 0011101000101100 =$$

$$+0.11101000101100$$

$$x_{16} = +0.908935546875$$

Expected value is $1.646 \times \cos(56.5^\circ) = 0.90848827782$. Our 16-bit representation provides precision till 3 rd decimal point.

GCD of two numbers

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

entity gcd_gauss_algorithm is
    Port (
        clk    : in  STD_LOGIC;
        rst_n   : in  STD_LOGIC;
        a       : in  STD_LOGIC_VECTOR(31 downto 0);
        b       : in  STD_LOGIC_VECTOR(31 downto 0);
        gcd     : out STD_LOGIC_VECTOR(31 downto 0)
    );
end gcd_gauss_algorithm;
```

architecture Behavioral of gcd_gauss_algorithm is

```
    signal temp_a, temp_b : std_logic_vector(31 downto 0);
    signal swap : std_logic;
```

begin

```
    temp_a <= a;
    temp_b <= b;
```

```
-- Ensure a >= b
```

```
swap <= '1' when (temp_a < temp_b) else '0';
```

```
(temp_a, temp_b) <= (temp_b, temp_a) when swap = '1' else (temp_a, temp_b);
```

```
-- Gauss algorithm loop
```

```
process(clk)
```

```
begin
```

```
    if rising_edge(clk) then
```

```
        if rst_n = '0' then
```

```
            temp_a <= (others => '0');
```

```
            temp_b <= (others => '0');
```

```
        else
```

```
            if temp_a >= temp_b then
```

```
                temp_a <= std_logic_vector(unsigned(temp_a) - unsigned(temp_b));
```

```
            end if;
```

```
        end if;
```

```
    end if;
```

```
end process;
```

```
gcd <= temp_a;
```

```
end Behavioral;
```