

# REPORT

## Thiết kế và thực hiện mạch tính hàm e-mũ

Ver 0.2

29/12/2025

	Họ và tên (Full name)	Mã SV (ID)	Đóng góp (Contribution)
Thành viên 1 (Member 1)	Đoàn Đức Mạnh	22022167	<ul style="list-style-type: none"><li>- Tìm hiểu thuật toán</li><li>- Thiết kế datapath</li><li>- Viết testbench</li><li>- Đưa ra ý tưởng cải tiến</li><li>- Viết báo cáo cho datapath và test bench</li></ul>
Thành viên 2 (Member 2)	Dương Trung Hiếu	22022208	<ul style="list-style-type: none"><li>- Tìm hiểu thuật toán</li><li>- Viết thiết kế control_unit, memory và top module</li><li>- Viết báo cáo các phần liên quan</li></ul>
Tên/Địa chỉ Repo trên Github hoặc Google Drive	<a href="https://github.com/JerryK4/ExpApprox.git">https://github.com/JerryK4/ExpApprox.git</a>		

### **Tóm tắt (Abstract - from 5 to 10 lines)**

Báo cáo này trình bày quy trình thiết kế và triển khai mạch tích hợp thực hiện phép tính hàm số mũ ( $e^t$ ) sử dụng ngôn ngữ mô tả phần cứng VHDL. Giải pháp được lựa chọn dựa trên phương pháp bảng tra cứu Look-Up Table LUT kết hợp với mô hình FSMD kết hợp giữa máy trạng thái FSM và Datapath. Ngoài ra có thêm 1 bộ nhớ ngoài để lưu giá trị tiền tính toán và kết quả sau tính toán của  $e^t$ . Toàn bộ kiến trúc được mô phỏng và kiểm chứng chức năng trên ModelSim nhằm đánh giá độ chính xác của giá trị đầu ra so với tính toán thực tế, sau đó được thực hiện tương tự trên Vivado. Kết quả mô phỏng trên cả 2 phần mềm là như nhau và cho giá trị kết quả gần sát với giá trị thực tế trong một khoảng giá trị đầu vào nhất định.

### **Từ khóa (Keywords)**

Hàm số mũ ( $e^t$ ), ModelSim, Vivado, máy trạng thái hữu hạn FSM, FSMD, Datapath, CORDIC Hyperbolic, Bảng tra cứu Look-Up Table (LUT), mô phỏng, Microprocessor,

## Document History

Version	Time	Revised by	Description
V0.1	27/04/2021	Nguyễn Kiên Hùng	Original Version
V0.2	29/12/2025	Dương Trung Hiếu Đoàn Đức Mạnh	Complete Version

### Hướng dẫn (Instructions)

Sinh viên điền vào báo cáo theo mẫu đính kèm. Sinh viên điền các mục:

- Thông tin sinh viên, mã số sinh viên
- Mục *Đóng Góp* điền các công việc đã làm tương ứng của từng sinh viên.
- Tên/Địa chỉ Repo trên Github

Ngoại trừ phần thông tin sinh viên, mã số sinh viên và tên/địa chỉ Repo trên Github ở đầu, sinh viên cần hoàn thành các phần nội dung (theo các mục đã được gợi ý – nhưng không hạn chế) trong phần báo cáo để mô tả các công việc nhóm đã thực hiện và các kết quả đã đạt được.

Sinh viên làm theo nhóm chỉ cần **1 sinh viên đại diện nộp 1 bản báo cáo, sửa tên file thành tên của các thành viên trong nhóm (viết có dấu).**

Sinh viên nộp lại báo cáo này trước khi tới trình bày kết quả, **muộn nhất trước ngày thi hết môn một ngày. Ngày thi, SV cần mang máy tính laptop và sản phẩm để chạy demo!**

**Lưu ý: Nghiêm cấm mọi hình thức copy bài (bao gồm cả report và mã nguồn) của nhau. Nếu phát hiện sự giống nhau giữa 2 bài thì tùy mức độ mà có thể sẽ bị trừ điểm hoặc chia lấy điểm trung bình làm điểm của project.**

# MỤC LỤC

<b>Document History .....</b>	<b>3</b>
<b>MỤC LỤC.....</b>	<b>5</b>
<b>1. Giới thiệu.....</b>	<b>7</b>
<i>Lưu ý khi triển khai phần cứng: .....</i>	<i>8</i>
<b>2. Yêu cầu .....</b>	<b>8</b>
2.1. <i>Yêu cầu đối với thiết kế: .....</i>	<i>8</i>
2.2. <i>Định nghĩa giao diện vào/ra: .....</i>	<i>9</i>
<b>3. Thuật toán.....</b>	<b>9</b>
3.1. <i>Thuật toán CORDIC trong chế độ Hyperbolic để tính hàm <math>e^x</math>. .....</i>	<i>9</i>
3.2. <i>Code C: .....</i>	<i>11</i>
<b>4. Thiết kế mức RTL .....</b>	<b>13</b>
4.1. <i>Mô hình máy FSMMD : .....</i>	<i>13</i>
4.2. <i>Đơn vị xử lý dữ liệu (Datapath) : .....</i>	<i>14</i>
4.3. <i>Đơn vị điều khiển (Control Unit) : .....</i>	<i>20</i>
4.4. <i>Sơ đồ khối tổng thể:.....</i>	<i>21</i>
<b>5. Mô hình hóa bằng VHDL .....</b>	<b>23</b>
5.1. <i>Các đoạn code trong Datapath .....</i>	<i>23</i>
5.1.1. MUX.....	23
5.1.2. Register.....	25
5.1.3. Comparator.....	27
5.1.4. Bộ dịch phải .....	29
5.1.5. Look-Up Table – LUT .....	30
5.1.6. Bộ tính toán .....	33
5.1.7. Datapath .....	34
5.2. <i>Control Unit : .....</i>	<i>38</i>
5.2.1. IIR và OIR.....	38
5.2.2. Memory – Bộ nhớ vào ra .....	40
5.2.3. Controller – Module điều khiển .....	42
5.2.4. Control_Unit: Kết hợp controller với các biến điều khiển địa chỉ Memory .....	45

5.2.5. Microprocessor: Vi xử lý cho việc tính Exponential.....	47
5.3. <i>Microprocessor + Memory = Top System</i> : .....	50
<b>6. Mô phỏng/thực thi và đánh giá: Testbench:.....</b>	<b>52</b>
<b>7. Đánh giá kết quả : .....</b>	<b>57</b>
<b>8. Kết luận:.....</b>	<b>59</b>
<b>Appendix A: Reference .....</b>	<b>60</b>
<b>[1].</b>	<b>60</b>
<b>Appendix B: VHDL Code.....</b>	<b>61</b>
<b>(đóng gói thành tệp nén và gửi kèm báo cáo) .....</b>	<b>61</b>
<b>Appendix C: .....</b>	<b>62</b>
<b>List of Figures .....</b>	<b>63</b>
<b>List of Tables.....</b>	<b>64</b>
<b>References .....</b>	<b>65</b>

## 1. Giới thiệu

*(Introduction to the motivation, Objectives, and main Contents of the project)*

**Mục tiêu:** Vận dụng các kiến thức, kỹ năng đã được học để thiết kế, mô phỏng và thực thi một mô-đun phần cứng, gọi là ExpApprox, thực hiện tính hàm e-mũ  $e = \exp(t) = e^t$ . Trong đó,  $t$  là một số thực bất kỳ.

1	Input: $t$ (số thực, có thể dương hoặc âm)
2	Output: $e \approx e^t$
3	Constants:
4	$N =$ số vòng lặp (độ chính xác, ví dụ $N = 15$ )
5	$LUT[i] = \operatorname{atanh}(2^{-i})$ for $i = 1..N$
6	$K = \prod_1^N \sqrt{1 - 2^{-2i}}$ for $i = 1..N$ (khoảng 1.2075)
7	Initialization:
8	$X \leftarrow 1.0/K$ // vector khởi tạo (chuẩn hóa)
9	$Y \leftarrow 0.0$
10	$Z \leftarrow t$ // góc cần xoay ( $x$ là đối số của $e^t$ )
11	Iteration:
12	for $i = 1$ to $N$ do
	if $Z \geq 0$ then
13	$X_{\text{next}} \leftarrow X + (Y * 2^{-i})$
14	$Y_{\text{next}} \leftarrow Y + (X * 2^{-i})$
15	$Z_{\text{next}} \leftarrow Z - LUT[i-1]$
16	else
17	$X_{\text{next}} \leftarrow X - (Y * 2^{-i})$
18	$Y_{\text{next}} \leftarrow Y - (X * 2^{-i})$
19	$Z_{\text{next}} \leftarrow Z + LUT[i-1]$
20	end if
21	$X \leftarrow X_{\text{next}}$
22	$Y \leftarrow Y_{\text{next}}$
23	$Z \leftarrow Z_{\text{next}}$
24	end for
25	Result:
26	$e^t \approx X + Y$

Hình 1. Thuật toán tính xấp xỉ hàm  $e = e^t$ .

**Ví dụ:** Giả sử  $(x = 1.0)$ ,  $(N = 5)$ . Sau 5 vòng lặp, ta nhận được  $(X + Y \approx 2.717)$ , rất gần với  $(e^1 = 2.71828)$ .

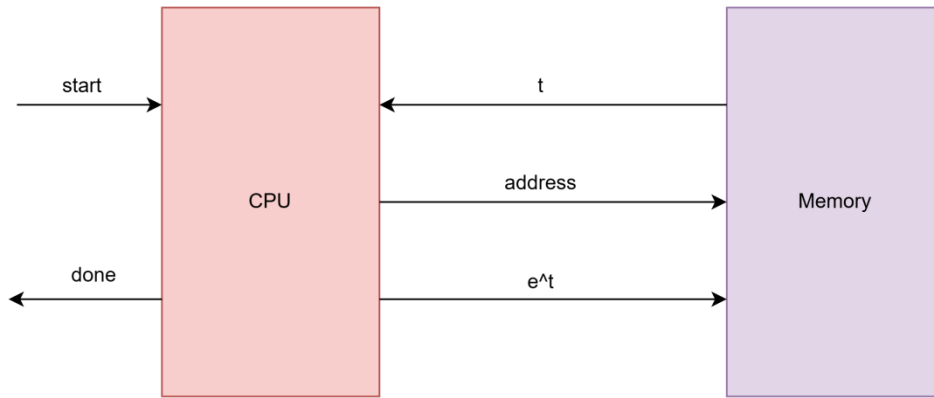
### Lưu ý khi triển khai phần cứng:

- Bảng LUT chỉ cần lưu 10–20 giá trị ( $\text{atanh}(2^{-(i)})$ ).
- Có thể **pipeline** các vòng lặp để tăng tốc.
- Dùng **fixed-point arithmetic** (ví dụ Q1.15 hoặc Q2.14 format). **Fixed-point arithmetic** (số học cố định) là cách **biểu diễn và tính toán số thực bằng số nguyên**, với **một vị trí dấu chấm thập phân cố định** trong biểu diễn nhị phân. Một giá trị được biểu diễn theo **định dạng Qm.n**, trong đó:
  - + m: số bit cho phần **nguyên** (bao gồm cả bit dấu nếu có)
  - + n: số bit cho phần **phân số**
  - + Tổng số bit =  $m + n$
- Nếu cần độ chính xác cao, ta nhân thêm hệ số (K) ở cuối

## 2. Yêu cầu

### 2.1. Yêu cầu đối với thiết kế:

- Khối ExpApprox có giao diện ghép nối tới CPU sao cho CPU có thể viết giá trị đầu vào cần tính (giá trị t) tới nó
- CPU kích hoạt quá trình tính toán của khối ExpApprox bằng các đặt tín hiệu  $\text{Start} = '1'$ .
- Sau khi quá trình tính  $\exp(t)$  hoàn thành, khối ExpApprox sẽ báo cho CPU biết bằng cách đặt tín hiệu  $\text{Done} = '1'$ ;
- ???



Hình 2. Giao diện ghép nối I/O.

## 2.2. Định nghĩa giao diện vào/ra:

TT	Port	Direction	Width	Meaning
1	Start	In	1 bit	Tín hiệu khởi động quá trình tính toán
2	Clock	In	1 bit	Tín hiệu xung nhịp
3	Reset	In	1 bit	Tín hiệu đặt lại hệ thống về trạng thái ban đầu
4	Done	Out	1 bit	Tín hiệu báo hiệu kết thúc quá trình tính toán

Bảng 1: Mô tả các tín hiệu vào ra.

## 3. Thuật toán

### 3.1. Thuật toán CORDIC trong chế độ Hyperbolic để tính hàm $e^t$ .

-Thông thường, khi tính toán  $e^t$  là một phép nhân liên tiếp. Nhưng trong toán học,  $e^t$  có mối quan hệ với các hàm Hyperbolic Sine(sinh) và Hyperbolic Cosine(cosh):

- Công thức:  $e^t = \cosh(t) + \sinh(t)$
- Ý nghĩa: Nếu trong không gian lượng giác tròn, một điểm trên đường tròn đơn vị ( $x^2 + y^2 = 1$ ) có tọa độ  $(\cos(\Theta), \sin(\Theta))$ , thì trong không gian Hyperbolic ( $x^2 - y^2 = 1$ ), một điểm trên nhánh Hyperbol có tọa độ  $(\cosh t, \sinh t)$ .

-Tư tưởng CORDIC: Thay phép nhân bằng dịch bit vì bộ nhân rất tốn tài nguyên phần cứng. Trong không gian Hyperbolic, phép quay một vector đi một góc  $\alpha$  được thực hiện bởi ma trận:

$$\begin{bmatrix} X_{i+1} \\ Y_{i+1} \end{bmatrix} = \begin{bmatrix} \cosh \alpha & \sinh \alpha \\ \sinh \alpha & \cosh \alpha \end{bmatrix} \begin{bmatrix} X_i \\ Y_i \end{bmatrix}$$

Rút  $\cosh \alpha$  ra ngoài :

$$\begin{bmatrix} X_{i+1} \\ Y_{i+1} \end{bmatrix} = \cosh \alpha \begin{bmatrix} 1 & \tanh \alpha \\ \tanh \alpha & 1 \end{bmatrix} \begin{bmatrix} X_i \\ Y_i \end{bmatrix}$$

Để tránh các phép nhân, CORDIC chọn các góc quay  $\alpha_i$  sao cho  $\tanh(\alpha_i) = 2^{-i}$  (một lũy thừa của 2). Khi đó, phép nhân với  $\tanh(\alpha_i)$  thực chất chỉ là phép dịch bit sang phải  $i$  lần.

-Để xoay góc  $t$  về 0, ta xoay thành nhiều bước nhỏ. Mỗi bước thứ  $i$ , ta xoay cố định một góc  $\alpha_i$ . Giá trị này được tính trước và lưu vào bảng Look-Up Table - LUT.

- Bước  $i = 1$ :  $\alpha_1 = \operatorname{atanh}(2^{-1}) = 0.5493$
- Bước  $i = 1$ :  $\alpha_1 = \operatorname{atanh}(2^{-1}) = 0.5493$
- ... và tiếp theo.

-Quy trình xoay dần đến 0. Ta có 3 biến được cập nhật liên tục:

- X: tương ứng với giá trị  $\cosh$  (tích lũy giá trị).
- Y: tương ứng với giá trị  $\sinh$  (tích lũy giá trị).
- Z: Góc còn lại cần phải xoay (ban đầu  $Z = t$ ).

Ở mỗi bước  $i$ , ta nhìn vào dấu của Z:

- Nếu  $Z > 0$ : Ta xoay theo chiều dương. Ta trừ Z đi một khoảng  $\alpha_i$  (để đưa Z về gần 0 hơn)
- Nếu  $Z < 0$ : Ta xoay theo chiều âm. Ta cộng thêm  $\alpha_i$  vào Z.

Sau nhiều vòng lặp, Z sẽ xấp xỉ bằng 0. Lúc này, các giá trị tích tụ trong X và Y chính là các thành phần của  $e^t$ .

-Sau  $n$  bước lặp, giá trị cuối cùng trong X và Y thực tế là:

- $X_n \approx K \cosh(t)$
- $Y_n \approx K \sinh(t)$

Trong đó K là hệ số kéo dẫn (Gain) xuất hiện do ta đã bỏ qua  $\cosh \alpha_i$  ở mỗi bước quay ma trận. K là một hằng số cố định xấp xỉ 1.20513.

-Để triệt tiêu K, ta có 2 cách:

- Nhân kết quả cuối cùng với  $1/K$ .
- Khởi tạo ban đầu: Thay vì đặt  $X_0 = 1$  ta đặt  $X_0 = 1/K$ . Như vậy kết quả cuối cùng sẽ tự động được chuẩn hóa.

Cuối cùng ta tính được  $e^t = X_n + Y_n = \cosh(t) + \sinh(t)$ .

### 3.2. Code C:

```
#include <stdio.h>
#include <math.h>

double LUT[16];
double LUT1[16];

int main(){
    int N = 15;
    for(int i=1; i<=N; i++){
        LUT1[i] = pow(2, -i);
    }
    for(int i=1; i<=N; i++){
        LUT[i] = atanh(LUT1[i]);
    }
    double X = 1.205136;
    double Y = 0;
    double Z; scanf("%lf", &Z);
    double X_next, Y_next, Z_next;
    for(int i=1; i<=N; i++){
        if(Z >= 0){
            X_next = X + (Y * LUT1[i]);
            Y_next = Y + (X * LUT1[i]);
            Z_next = Z - LUT[i];
        }else{
            X_next = X - (Y * LUT1[i]);
            Y_next = Y - (X * LUT1[i]);
            Z_next = Z + LUT[i];
        }
        X = X_next; Y = Y_next; Z = Z_next;
    }
    printf("%lf", X+Y);
}
```

Dựa vào công thức đã chứng minh ở phần trước, ta sẽ xây dựng thuật toán bao gồm:

- 2 bảng LUT1 và LUT dùng để chứa các giá trị  $2^{-k}$  và  $\operatorname{atanh}(2^{-k})$  với  $1 \leq k \leq 15$ , giới hạn trên có thể tăng lên để thuật toán chính xác hơn
- 2 vòng lặp đầu để gán giá trị cho LUT1 và LUT
- Khai báo các biến X,Y là thành phần của vector xoay, Z là biến tích lũy góc với
  - $X = 1.205136$ , là 1 hằng số vì ta chọn lặp 15 lần, được tính ra từ công thức:

$$\frac{1}{\prod_{i=1}^{15} \sqrt{1 - 2^{-2i}}}$$

- $Y = 0$
  - Z là giá trị t để tính  $e^t$ , được nhập từ bàn phím
- Sau đó, ta bước vào vòng lặp với i đi từ 1 đến 15 để thực hiện thuật toán cập nhật của Hyperbolic CORDIC
  - Đầu tiên so sánh Z với 0 để biết hướng điều chỉnh sao cho Z tiến gần về không

- Với  $Z > 0$ , ta cập nhật

$$X = X + Y * 2^{-i}$$

$$Y = Y + X * 2^{-i}$$

$$Z = Z - \operatorname{atanh}(2^{-i})$$

- Với  $Z < 0$ , ta cập nhật

$$X = X - Y * 2^{-i}$$

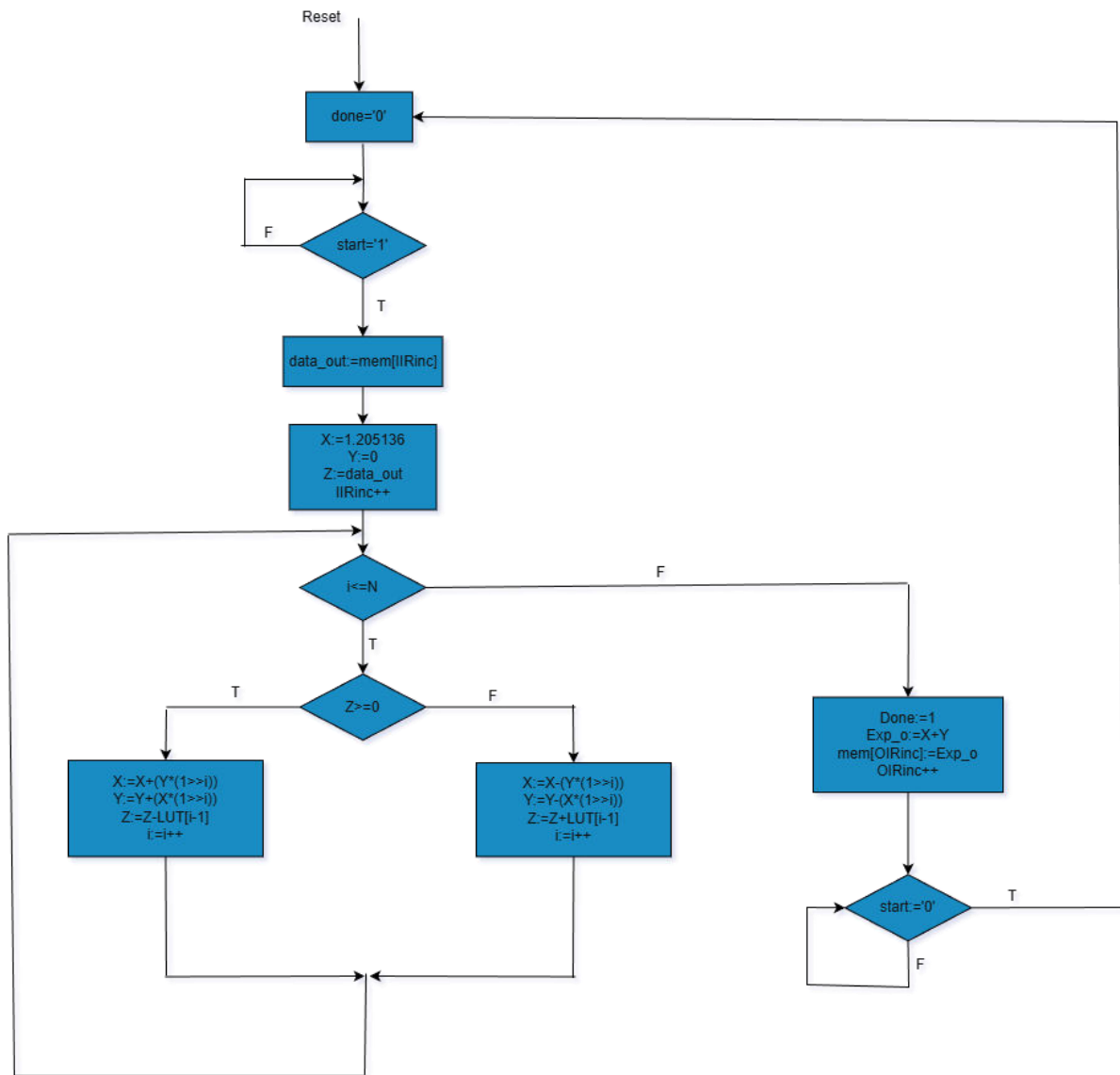
$$Y = Y - X * 2^{-i}$$

$$Z = Z + \operatorname{atanh}(2^{-i})$$

- Cuối cùng, ta in ra kết quả cuối là giá trị tổng của X và Y

## 4. Thiết kế mức RTL

### 4.1. Mô hình máy FSMD :



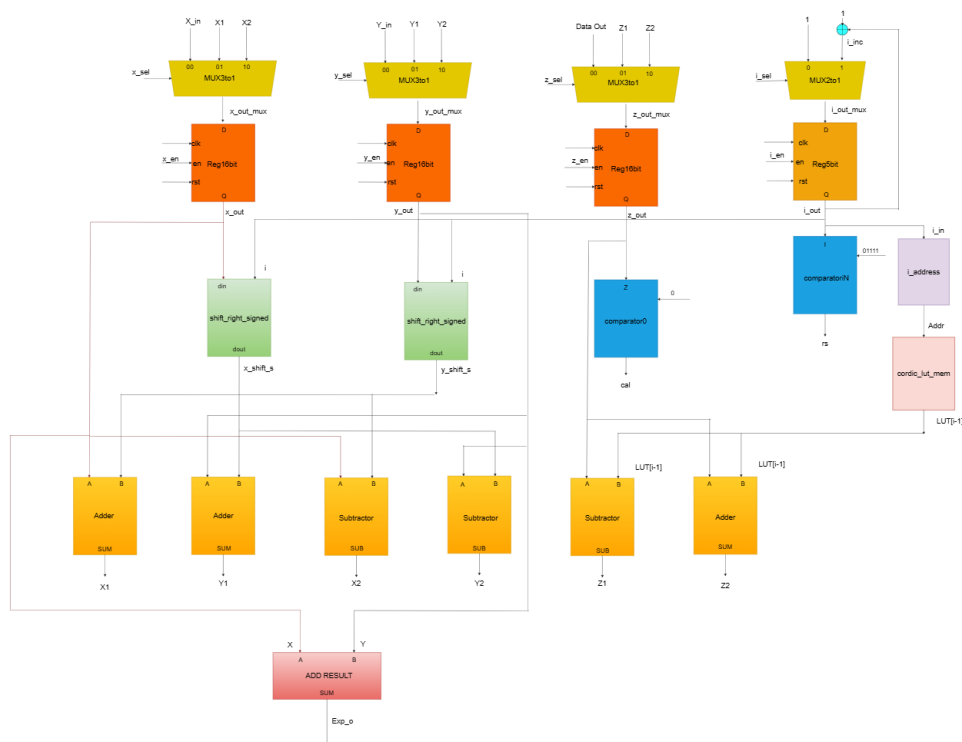
**Hình 3: Mô hình máy FSMD.**

Ta xây dựng mô hình FSMD như sau:

- Máy trạng thái bắt đầu với khối `done='0'` để chốt khi nào dữ liệu sẵn sàng.
- Sau đó nó idle ở khối so sánh `start='1'` để biết khi nào có thể bắt đầu thực thi chương trình
- Khối tiếp theo mô tả bước lấy giá trị đầu vào mong muốn  $x$  của  $e^x$  từ memory ở địa chỉ `IIRinc`
- Sau đó, các giá trị  $X, Y, Z, I$  sẽ được khởi tạo giá trị ban đầu
- Thuật toán tiến tới vòng lặp, ở thuật toán này ta chọn  $N=15$ , nghĩa là  $i$  từ 1->15

- So sánh Z với 0, ta đi 2 hướng cập nhật cho các giá trị X,Y,Z theo thuật toán Hyperbolic Cordic
- Cập nhật các giá trị X,Y,Z và tăng I lên 1.
- Đặt giá trị Done='0', tính ra đầu ra và đưa vào mem[OIRinc], tăng biến đếm OIR cho các chu kì tới
- Kiểm tra xem start='0' hay không để biết đã sẵn sàng cho chu kì mới chưa, tránh bị lỗi dữ liệu

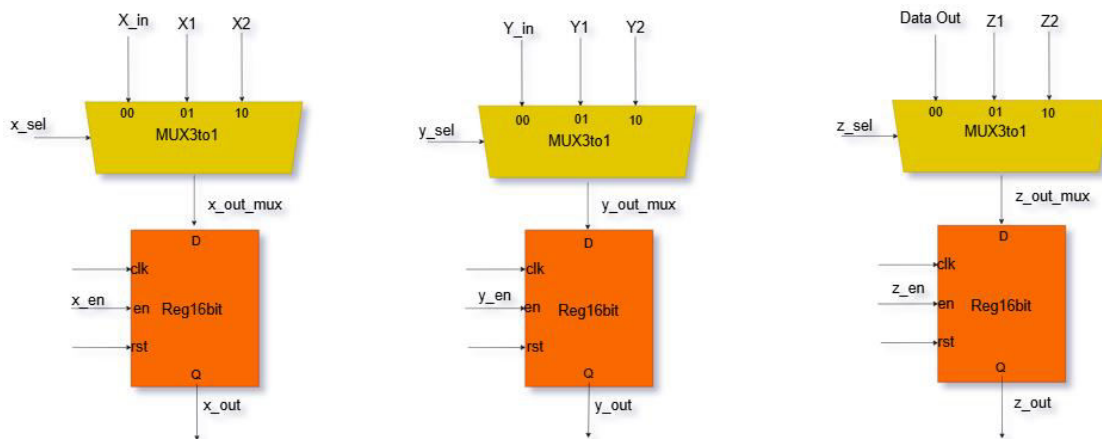
#### 4.2. Đơn vị xử lý dữ liệu (Datapath) :



**Hình 4: Cấu trúc của đơn vị xử lý dữ liệu Datapath.**

Ta sẽ phân tích từng thành phần trong Datapath:

**-Khối lựa chọn đầu vào và thanh ghi lưu trữ dữ liệu tạm thời**



Cấu tạo của khối gồm 2 thành phần là bộ Multiplexer Mux 3to1 và bộ thanh ghi 16 bit :

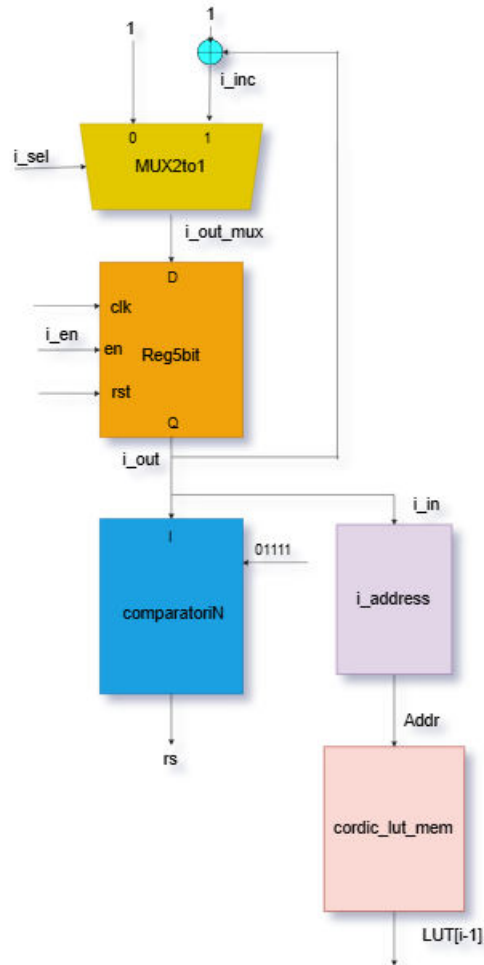
+Bộ MUX3to1 quyết định các giá trị đầu vào cho thanh ghi. Có 3 trường hợp xảy ra, trường hợp đầu tiên là giá trị khởi tạo  $X_{in}$ ,  $Y_{in}$ ,  $Z_{in}$  ( chính là giá trị  $t$  trong  $e^t$ ). , 2 trường hợp còn lại được lấy từ giá trị tính toán ở tầng dưới:

- Ngõ vào 00: Nhận giá trị khởi tạo ban đầu (  $X_{in}$ ,  $Y_{in}$ ,  $Z_{in}$ ). Đây là nơi nạp dữ liệu vào mạch khi bắt đầu một phép tính  $e^t$  mới.
- Ngõ vào 01 và 10: Nhận các giá trị phản hồi( $X1$ ,  $X2$ ,  $Y1$ ,  $Y2$ ,  $Z1$ ,  $Z2$ ) từ kết quả của các bộ tính toán (cộng, trừ) ở tầng dưới.
- Tín hiệu điều khiển (  $x_{sel}$ ,  $y_{sel}$ ,  $z_{sel}$  ) có độ dài 2bit để có thể lựa chọn được 3 ngõ vào. Các tín hiệu này do bộ điều khiển quyết định xem mạch đang ở giai đoạn “Nạp dữ liệu mới” hay “Đang tính toán”.

+Thanh ghi Reg16bit (Register) lưu trữ giá trị tạm thời của X, Y, Z :

- D (Data in) : Nhận giá trị từ đầu ra của bộ MUX
- Q (Data out): Xuất giá trị hiện tại ra các khối dịch bit và bộ cộng bên dưới.
- Chân điều khiển:
  - clk : Xung đồng hồ điều khiển nhịp tính toán.
  - en(Enable): Cho phép cập nhật giá trị mới vào thanh ghi.
  - rst(Reset): Đưa giá trị thanh ghi về 0.

**-Khối quản lý vòng lặp và lấy giá trị bằng LUT**



+Bộ MUX 2to1 có chức năng chọn giá trị để nạp vào bộ đếm vòng lặp.

- Ngõ vào I0: Dùng để khởi tạo vòng lặp, giá trị khởi tạo  $i = 1$ .
- Ngõ vào I1: Nhận giá trị  $i + 1$  từ bộ tăng (inc1) để chuyển sang vòng lặp tiếp theo.
- Tín hiệu điều khiển  $i\_sel$ : Quyết định việc lựa chọn đầu vào bộ Mux, được điều khiển bởi bộ controller

+Thanh ghi 5-bit (reg5bit) có chức năng lưu trữ chỉ số vòng lặp hiện tại( $i$ ) , đầu vào của thanh ghi nhận giá trị đầu ra của bộ Mux:

- Chân en(Enable): Chỉ khi bộ điều khiển cho phép, giá trị  $i$  mới được cập nhật.
- Chân rst(Reset): Đưa bộ đếm về 0 khi hệ thống khởi động lại.
- Độ rộng 5 bit cho phép đếm tối đa  $2^5 = 32$  giá trị phù hợp với độ chính xác 16 bit dữ liệu.

+Bộ tăng chỉ số(inc1): Thực hiện phép toán  $i\_next = i\_current + 1$ .

- Giá trị đầu vào được lấy từ đầu ra của thanh ghi 5 bit.
- Kết quả được đưa vào đầu I1 của bộ Mux để chờ nạp cho chu kỳ tiếp theo.

+Bộ so sánh với giá trị  $N = 15$  - số vòng lặp (comparatorN)

- Kiểm tra xem đã hoàn thành đủ số vòng lặp cần thiết chưa.
- Giá trị đầu vào chính là kết quả đầu ra của thanh ghi 5 bit. Giá trị kết quả đầu ra là tín hiệu rs.
- So sánh giá trị  $i$  hiện tại xem còn thỏa mãn điều kiện  $i \leq N$  hay không. Nếu  $i \leq N$ , tín hiệu đầu ra  $rs = 1$  (tiếp tục chạy), nếu  $rs = 0$  báo hiệu đã hoàn thành việc tính toán. Tín hiệu rs này sẽ được gửi đến Control Unit để điều khiển các trạng thái.

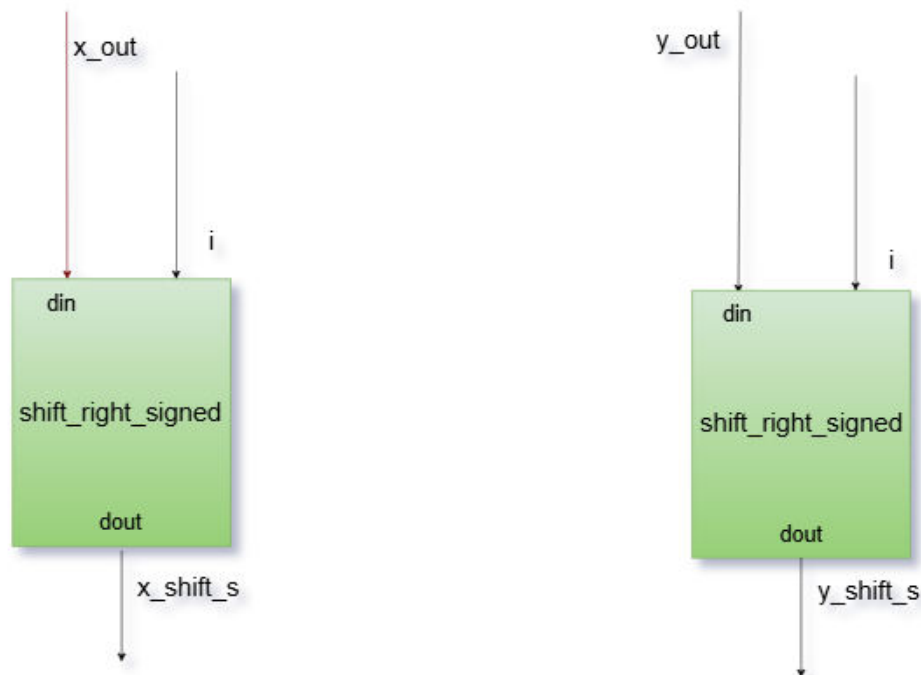
+Bộ chuyển đổi địa chỉ( $i\_address$ ):

- Chuyển đổi chỉ số vòng lặp  $i$  thành địa chỉ bộ cordic\_lut\_mem.
- Trong toán học CORDIC Hyperbolic, các bước tính toán bắt đầu từ  $i = 1, 2, 3 \dots$ . Tuy nhiên bộ nhớ cordic\_lut\_mem lại bắt đầu từ địa chỉ 0. Do đó, tại vòng lặp thứ  $i$  sẽ lấy dữ liệu tại vị trí  $i-1$ . Đầu ra của bộ chuyển đổi chính là địa chỉ đầu vào cho bộ lưu trữ bảng LUT.

+Bộ lưu trữ bảng Look-Up Table LUT

- Lưu trữ các giá trị hằng số góc Hyperbolic  $\operatorname{atanh}(2^{-i})$ .
- Các giá trị dữ liệu được lưu dưới định dạng Q3.13 với 3 bit cho phần nguyên và 13 bit cho phần phân số để đảm bảo độ chính xác khi tính toán số thực trên phần cứng.

**-Khối dịch bit có dấu**



- Trong thuật toán CORDIC Hyperbolic yêu cầu thực hiện phép tính:

$$Y * 2^{(-i)} \text{ và } X * 2^{(-i)}$$

- Thay vì sử dụng bộ nhân (Multiplier) - rất tốn tài nguyên phần cứng và làm chậm tốc độ mạch - khối dịch bit này thực hiện phép nhân đó bằng cách dịch chuyển các bit sang bên phải  $i$  lần.
- Trong bộ dịch đảm bảo bảo toàn dấu cho phép toán. Vì dữ liệu tọa độ  $X, Y$  trong CORDIC có thể là số âm ( biểu diễn ở dạng bù 2 ), khối dịch đã đảm bảo:
  - Bảo toàn dấu: Khi dịch bit sang phải, thay vì đẩy số 0 vào các vị trí trống bên trái, khối này sẽ đẩy giá trị bit dấu (MSB) vào, cơ chế này dựa vào hàm `shift_right` trong thư viện `ieee.numeric_std.all`.
  - Đảm bảo số âm sau khi dịch vẫn là số âm và giữ nguyên giá trị đại số chính xác.
- Đầu vào của bộ dịch là kết quả đầu ra của 2 thanh ghi 16 bit , đầu ra là kết quả sau khi dịch phải  $i$  lần.

#### -Khối tính toán và quyết định chiều xoay



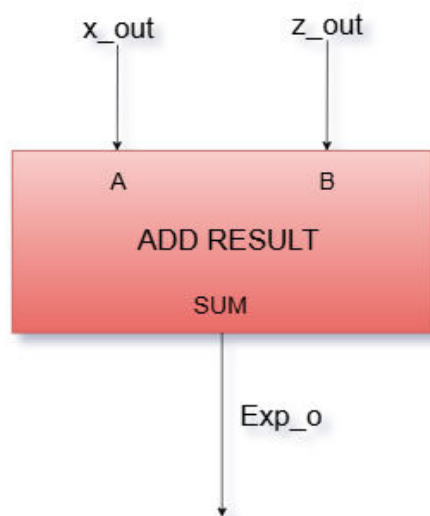
#### +Bộ so sánh dấu

- Đây là bộ điều khiển hướng xoay của vector. Nó kiểm tra xem giá trị  $Z$  hiện tại là dương hay âm :
  - Nếu  $Z \geq 0 \Rightarrow cal = 1$  , nghĩa là ta cần xoay theo chiều âm để giảm  $Z$  về 0.
  - Nếu  $Z < 0 \Rightarrow cal = 0$  , nghĩa là ta cần xoay theo chiều dương để tăng  $Z$  về 0.
- Tín hiệu `cal` sẽ được gửi về về Control Unit để quyết định các tín hiệu, trạng thái điều khiển.

#### +Bộ cộng và bộ trừ (adder và subtractor)

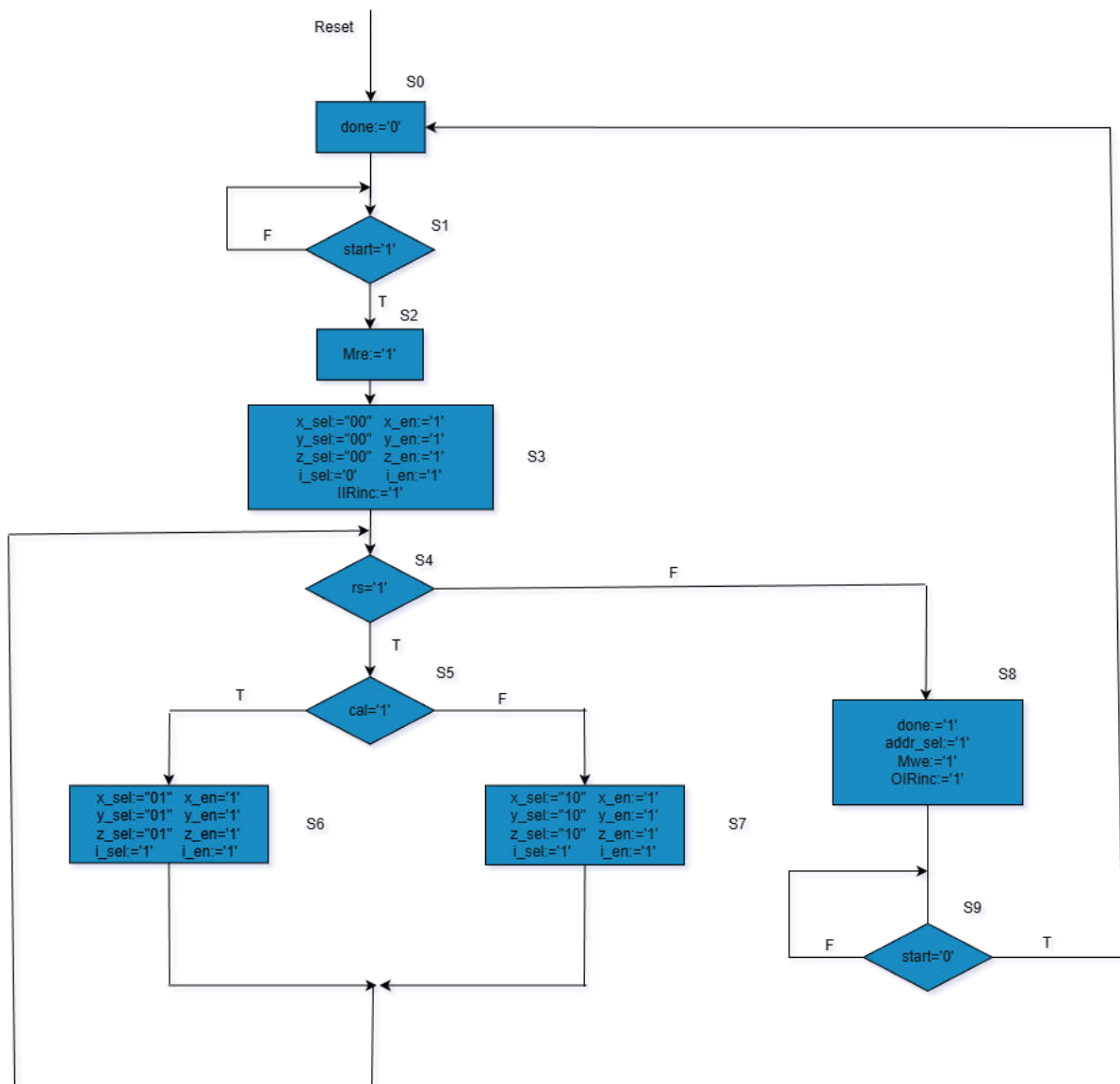
- Mạch sử dụng song song các bộ cộng và bộ trừ để chuẩn bị sẵn các phương án kết quả, giúp tối ưu tốc độ xử lý.
- Nhánh tính toán cho X:
  - Adder(X1): Thực hiện  $X + (Y \gg i)$ . Kết quả này được chọn nếu  $Z \geq 0$ .
  - Subtractor(X2): Thực hiện  $X - (Y \gg i)$ . Kết quả này được chọn nếu  $Z < 0$ .
- Nhánh tính toán cho Y:
  - Adder(Y1): Thực hiện  $Y + (X \gg i)$ . Kết quả này được chọn nếu  $Z \geq 0$ .
  - Subtractor(Y2): Thực hiện  $Y - (X \gg i)$ . Kết quả này được chọn nếu  $Z < 0$ .
- Nhánh tính toán cho Z:
  - Subtractor(Z1): Thực hiện  $Z - LUT[i-1]$ . Kết quả này được chọn nếu  $Z \geq 0$ .
  - Adder(Z2): Thực hiện  $Z + LUT[i-1]$ . Kết quả này được chọn nếu  $Z < 0$ .

#### **-Khối tính toán kết quả cuối cùng**



- Khối này thực hiện 1 nhiệm vụ duy nhất là cộng 2 thành phần X và Y sau khi quá trình xoay CORDIC kết thúc.
- Đầu vào A,B: Nhận giá trị đầu ra từ các thanh ghi X ( $x\_out$ ) và Y ( $y\_out$ ) sau khi đã thực hiện đủ N vòng lặp.
- Đầu ra SUM( $Exp\_o$ ): Chính là kết quả cuối cùng của phép tính  $e^t$ .

#### 4.3. Đơn vị điều khiển (Control Unit) :



**Hình 5: Máy FSM của đơn vị điều khiển.**

Ta xây dựng mô hình FSD như sau:

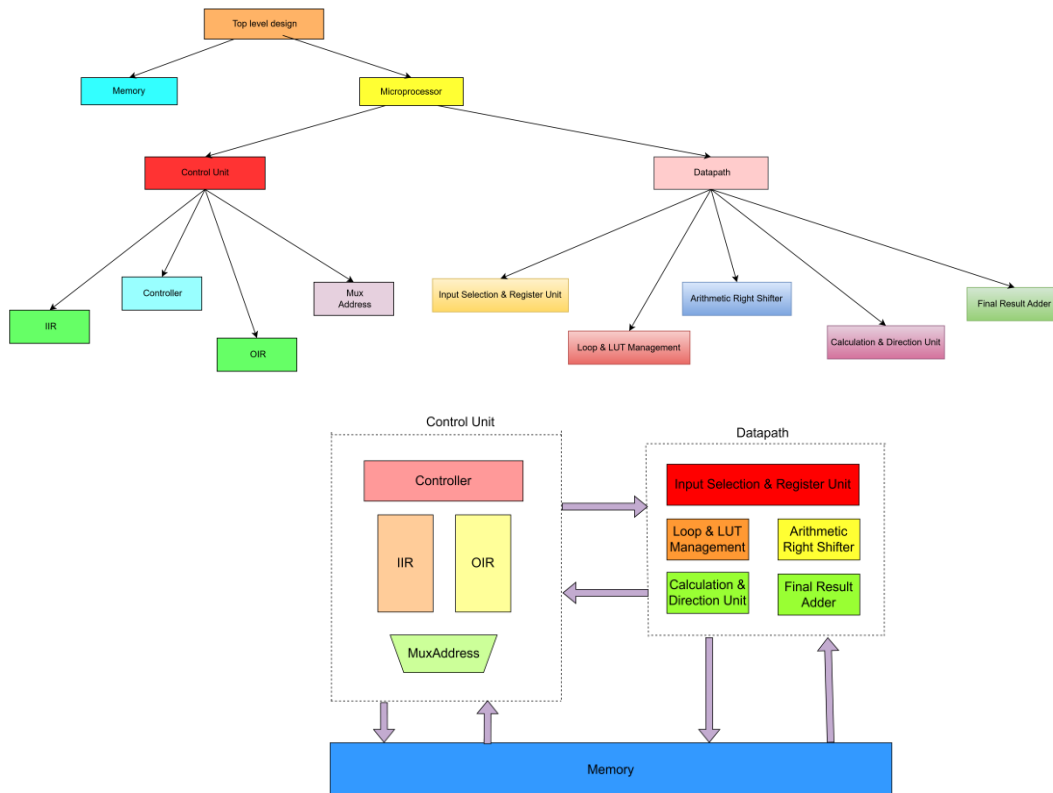
- Máy trạng thái bắt đầu ở trạng thái S0, khi ta ấn nút reset hoặc mới khởi động mạch, khi này tất cả thanh ghi sẽ bị reset và tín hiệu done sẽ bị gán thành 0 (chốt khi bắt đầu sẽ chờ lỗi khi tính toán lại và sẽ cho chính xác, sau đó chuyển qua S1
- Ở trạng thái S1, kiểm tra xem tín hiệu start đã được kéo lên 1 hay chưa, nếu rồi thì bắt đầu quá trình tính toán và qua S2, nếu không sẽ idle ở S1
- Qua trạng thái S2, tín hiệu Mre được set là 1 để đọc dữ liệu vào từ memory, qua trạng thái S3.
- Khi ở trạng thái S3, các tín hiệu x\_sel, y\_sel, z\_sel đều bằng 00, i\_sel = 0, nghĩa là mux chọn đầu vào là các giá trị khởi tại, các tín hiệu x\_en, y\_en, z\_en, i\_en được set là 1 để

thanh ghi chốt dữ liệu. Tín hiệu IIRinc=1 để tăng địa chỉ đọc ở memory lên 1 cho phép tính sau. Chuyển sang S4.

- Ở trạng thái S4, kiểm tra tín hiệu rs của phép so sánh i với 15, nếu rs='1' nghĩa là  $i \leq 15$  thì vào vòng lặp, qua trạng thái S6, ngược lại qua S8 để chốt kết quả
- Ở trạng thái S5, ta xem tín hiệu cal (tín hiệu so sánh Z với 0) là 0 hoặc 1 để chuyển sang S6 hoặc S7.
- S6 và S7 gần giống nhau, ta set  $x\_sel, y\_sel, z\_sel = 01$  nếu là S6 để chọn đầu vào là các phép cập nhật khi  $Z > 0$ ,  $x\_sel, y\_sel, z\_sel = 10$  nếu là S7 để chọn đầu vào là các phép cập nhật khi  $Z < 0$ , các tín hiệu còn lại giống nhau ở cả 2 trạng thái là  $i\_sel = 1$  để chọn i sau khi tăng và các tín hiệu  $x\_en, y\_en, z\_en, i\_en$  cho phép chốt dữ liệu. Chuyển về S4.
- Ở trạng thái S8, tín hiệu Done được nâng lên 1 để chốt dữ liệu đầu ra, các tín hiệu  $addr\_sel, Mwe = 1$  để khởi động quá trình ghi ra bộ nhớ, OIRinc tăng thêm 1 để trở tới ô nhớ lưu kết quả tiếp theo. Chuyển sang trạng thái S9
- Ở trạng thái S9, kiểm tra tín hiệu start đã bằng 0 hay chưa để chốt kết thúc quy trình, tránh việc tín hiệu start giữ ở 1 làm nhiều chu kỳ hoạt động tiếp theo. Nếu start = 0 thì chuyển qua S0 để bắt đầu chu kỳ mới, nếu start = 1 thì idle ở S9.

#### 4.4. Sơ đồ khối tổng thể:





**Hình 5: Sơ đồ khối tổng thể của bộ tính ảnh tích phân.**

Khối tính toán sẽ bao gồm 3 thành phần chính là

- Control Unit
- Datapath
- Memory

Trong đó Control Unit và Datapath sẽ tạo thành Microprocessor.

- Trong Control Unit có bao gồm các khối
  - Controller : Khối điều khiển sẽ quyết định trạng thái của máy.
  - IIR: Thanh ghi chứa địa chỉ của mảng đầu vào.
  - OIR: Thanh ghi chứa địa chỉ của mảng đầu ra.
  - MuxAddress: Khối lựa chọn địa chỉ được lưu trên phần bộ nhớ dành cho đầu vào (OIR) hay phần bộ nhớ dành cho đầu ra (IIR).
- Trong Datapath bao gồm các khối:
  - Input Selection & Register Unit: Khối lựa chọn đầu vào và thanh ghi lưu trữ dữ liệu tạm thời.
  - Loop & LUT Management: Khối quản lý vòng lặp và lấy giá trị bảng LUT
  - Arithmetic Right Shifter: Khối dịch bit có dấu.

- Calculation & Direction Unit: Khối tính toán và quyết định chiều xoay.
- Final Result Adder: Khối tính toán giá trị cuối cùng.
- Memory sẽ có các đầu vào là Address, DataIn, DataOut, Mre và Mwe.

## 5. Mô hình hóa bằng VHDL

### 5.1. Các đoạn code trong Datapath

#### 5.1.1. MUX

Bộ mux3to1\_16bit :

```
library ieee;
use ieee.std_logic_1164.all;

entity mux3to1 is
  generic(
    WIDTH : integer := 16
  );
  port(
    sel : in std_logic_vector(1 downto 0);
    I0 : in std_logic_vector(WIDTH-1 downto 0);
    I1 : in std_logic_vector(WIDTH-1 downto 0);
    I2 : in std_logic_vector(WIDTH-1 downto 0);
    O : out std_logic_vector(WIDTH-1 downto 0)
  );
end mux3to1;

architecture dataflow of mux3to1 is
begin
  with sel select
    O <= I0 when "00",
      I1 when "01",
      I2 when "10",
      (others => '0') when others;
end dataflow;
```

Đây là module bộ hợp kênh với lối vào 16bit :

- Chức năng: Cho phép chọn 1 trong 3 đường dữ liệu đầu vào(I0, I1, I2) để đưa tới đầu ra (O) dựa trên giá trị của tín hiệu điều khiển (sel)
- Generic: WIDTH: integer := 16: Đây là 1 tham số linh hoạt. Mặc định là 16 bit (khớp với độ rộng dữ liệu X,Y,Z ) nhưng có thể đổi thành 32 bit hoặc 8 bit mà không cần sửa lại code logic bên trong.
- Các cổng (Ports):

- sel (2 bit): Tín hiệu điều khiển. Vì có 3 đầu vào, chúng ta cần ít nhất 2 bit (có thể biểu diễn 4 trạng thái từ 00 đến 11).
- I0, I1, I2 (Dữ liệu vào): Có độ rộng theo biến WIDTH. Nhận dữ liệu từ bên ngoài hoặc từ các khối tính toán phản hồi.
- O (Dữ liệu ra): Có cùng độ rộng với đầu vào, là kết quả của phép chọn.
- Logic:
  - with ... select: Đây là một câu lệnh gán tín hiệu đồng thời (concurrent). Nó hoạt động như một bảng tra cứu tức thời trên phần cứng.
  - when "00", "01", "10": Tương ứng với việc chọn các kênh dữ liệu.
  - when others:
    - Vì tín hiệu sel có 2 bit, về mặt lý thuyết sẽ có trạng thái "11".
    - Hơn nữa, trong mô phỏng, tín hiệu có thể rơi vào các trạng thái lỗi như 'U' (chưa khởi tạo) hoặc 'X' (không xác định).
    - Dòng (others => '0') đảm bảo rằng nếu có bất kỳ trạng thái lạ nào xảy ra, đầu ra sẽ được ép về 0, giúp mạch hoạt động an toàn và tránh các lỗi lờ lững (latches).

### Bộ Mux2to1

```
library ieee;
use ieee.std_logic_1164.all;

entity mux2to1 is
  port(
    sel: in std_logic;
    I0: in std_logic_vector(4 downto 0);
    I1: in std_logic_vector(4 downto 0);
    O: out std_logic_vector(4 downto 0)
  );
end mux2to1;

architecture dataflow of mux2to1 is
begin
  O <= I1 when sel = '1' else I0;
end dataflow;
```

Đây là module bộ hợp kênh với lối vào 5 bit:

- Chức năng: Cho phép chọn 1 trong 2 đường dữ liệu đầu vào ( $I_0, I_1$ ) để đưa tới đầu ra ( $O$ ) dựa trên giá trị của tín hiệu điều khiển ( $sel$ ). Trong hệ thống CORDIC này, khối này cụ thể dùng để quản lý chỉ số vòng lặp  $i$ , chọn giữa giá trị khởi tạo và giá trị kế tiếp.
- Các cổng (Ports):
  - sel (1 bit): Tín hiệu điều khiển kiểu std\_logic. Vì chỉ có 2 đầu vào nên chỉ cần 1 bit để lựa chọn (mức '0' hoặc '1').
  - I0, I1 (Dữ liệu vào): Có độ rộng cố định là 5 bit (4 downto 0). Độ rộng này đủ để biểu diễn các số từ 0 đến 31, phù hợp với số lượng vòng lặp tối đa của bộ đếm  $i$

- O (Dữ liệu ra): Có độ rộng 5 bit, là giá trị được chọn để đưa vào thanh ghi lưu trữ vòng lặp (Reg5bit).
- Logic:
  - when ... else: Đây là câu lệnh gán tín hiệu có điều kiện (conditional signal assignment). Nó tạo ra một mạch logic tổ hợp cho phép chuyển mạch dữ liệu tức thời.
  - sel = '1': Khi tín hiệu chọn ở mức cao, đầu ra O sẽ được kết nối với nhánh dữ liệu  $I_1$  (thường là giá trị  $i$  đã được tăng tiến).

### 5.1.2. Register

Bộ reg16bit

```
library ieee;
use ieee.std_logic_1164.all;

entity reg16bit is
  port(
    clk: in std_logic;
    D: in std_logic_vector(15 downto 0);
    en: in std_logic;
    rst: in std_logic;
    Q: out std_logic_vector(15 downto 0)
  );
end reg16bit;

architecture behav of reg16bit is
begin
  process(clk, rst)
  begin
    if rst = '1' then
      Q <= (others => '0');
    elsif rising_edge(clk) then
      if en = '1' then
        Q <= D;
      end if;
    end if;
  end process;
end behav;
```

Đây là module thanh ghi lưu trữ dữ liệu 16-bit:

- Chức năng: Lưu trữ và duy trì giá trị tạm thời của các biến cốt lõi (X, Y hoặc Z) sau mỗi chu kỳ tính toán. Đây là thành phần quan trọng nhất để thực hiện tính chất "đệ quy" của thuật toán CORDIC (lấy kết quả vòng lặp trước làm đầu vào cho vòng lặp sau).
- Các cổng (Ports):
  - clk (1 bit): Xung đồng hồ hệ thống. Mọi hoạt động nạp dữ liệu đều được đồng bộ theo cạnh lên của tín hiệu này.

- rst (1 bit): Tín hiệu Reset không đồng bộ (Asynchronous). Khi chân này lên mức cao, thanh ghi bị xóa về 0 ngay lập tức mà không cần chờ xung clock.
  - en (1 bit): Tín hiệu cho phép nạp dữ liệu (Enable). Nó giúp bộ điều khiển (FSM) quản lý việc khi nào cần cập nhật giá trị mới và khi nào cần giữ nguyên kết quả cuối cùng.
  - D (16 bit): Dữ liệu đầu vào. Nhận giá trị từ bộ hợp kênh (MUX 3to1).
  - Q (16 bit): Dữ liệu đầu ra. Cung cấp giá trị hiện tại cho khối dịch bit và khối tính toán cộng/trừ.
- Logic:
    - process(clk, rst): Đây là một tiến trình nhảy theo cả xung nhịp và tín hiệu reset. Việc đưa rst vào danh sách nhảy (sensitivity list) là điều kiện bắt buộc để hiện thực hóa cơ chế reset không đồng bộ.
    - rst = '1': Đây là điều kiện có ưu tiên cao nhất. Khi hệ thống yêu cầu reset, đầu ra Q được ép về 0 (others => '0')
    - rising\_edge(clk): Chỉ khi không có lệnh reset và có một cạnh lên của xung đồng hồ, các logic bên trong mới được thực thi.
    - en = '1': Nếu en kích hoạt, giá trị tại cổng D sẽ được nạp vào thanh ghi và xuất hiện tại cổng Q. Nếu en = '0', thanh ghi sẽ giữ nguyên giá trị cũ (chức năng lưu trữ dữ liệu). Điều này ngăn chặn việc dữ liệu bị thay đổi ngoài ý muốn khi thuật toán đã tính toán xong.

#### Bộ reg5bit

```
library ieee;
use ieee.std_logic_1164.all;

entity reg5bit is
  port(
    clk: in std_logic;
    D: in std_logic_vector(4 downto 0);
    en: in std_logic;
    rst: in std_logic;
    Q: out std_logic_vector(4 downto 0)
  );
end reg5bit;

architecture behav of reg5bit is
begin
  process(clk, rst)
  begin
    if rst = '1' then
      Q <= (others => '0');
    elsif rising_edge(clk) then
      if en = '1' then
        Q <= D;
      end if;
    end if;
  end process;
end architecture;
```

```

end if;
end process;
end behav;

```

Đây là module thanh ghi lưu trữ dữ liệu 5-bit:

- Chức năng: Lưu trữ và duy trì chỉ số vòng lặp hiện tại ( $i$ ) của thuật toán CORDIC. Giá trị này cực kỳ quan trọng vì nó điều khiển khoảng cách dịch bit trong khối Shifter và xác định địa chỉ để tra cứu bảng LUT.
- Các cổng (Ports):
  - clk (1 bit): Xung đồng hồ điều khiển nhịp đếm của vòng lặp.
  - rst (1 bit): Tín hiệu Reset không đồng bộ. Khi kích hoạt, đưa chỉ số vòng lặp về 0 ngay lập tức để chuẩn bị cho một phép tính mới.
  - en (1 bit): Tín hiệu cho phép cập nhật (Enable). Chỉ khi en = '1', chỉ số vòng lặp mới tăng lên giá trị kế tiếp. Điều này giúp mạch dừng lại đúng lúc khi đã thực hiện đủ số vòng lặp cần thiết.
  - D (5 bit): Dữ liệu vào từ bộ hợp kênh mux2to1. Với 5 bit, thanh ghi có thể lưu trữ giá trị từ 0 đến 31, đủ để quản lý số vòng lặp cho độ chính xác cao.
  - Q (5 bit): Dữ liệu ra. Giá trị này được đưa đến bộ tăng tiến ( $i + 1$ ), bộ so sánh kết thúc, khối dịch bit và bộ tính toán địa chỉ LUT.
- Logic:
  - process(clk, rst): Tiến trình này giám sát đồng thời xung nhịp và tín hiệu reset.
  - Reset không đồng bộ (if rst = '1'): Có ưu tiên cao nhất. Ngay khi có tín hiệu reset, giá trị  $i$  bị xóa về 0 mà không cần đợi cạnh lên của clock. Điều này đảm bảo hệ thống có thể dừng khẩn cấp hoặc khởi động lại tức thì.
  - Cập nhật đồng bộ (elsif rising\_edge(clk)): Khi không reset, tại mỗi cạnh lên của clock, mạch kiểm tra chân en.
  - Cơ chế lưu giữ (if en = '1'): Nếu được cho phép, giá trị  $D$  (chỉ số vòng lặp mới) sẽ được chốt vào thanh ghi. Nếu en = '0', thanh ghi giữ nguyên giá trị cũ, giúp mạch ổn định trạng thái khi quá trình tính toán  $e^t$  hoàn tất.

### 5.1.3. Comparator

Bộ comparatorN

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity comparatorN is
  port(
    i : in std_logic_vector(4 downto 0);
    N : in std_logic_vector(4 downto 0);

```

```

    rs : out std_logic
);
end comparatorN;

architecture dataflow of comparatorN is
begin
    rs <= '1' when unsigned(i) <= unsigned(N) else '0';
end dataflow;

```

Đây là module bộ so sánh để xác định điều kiện dừng thuật toán:

- Chức năng: Kiểm tra xem quá trình tính toán CORDIC đã thực hiện đủ số vòng lặp cần thiết hay chưa.
- Các cổng (Ports):
  - i (5 bit): Chỉ số vòng lặp hiện tại lấy từ thanh ghi reg5bit.
  - N (5 bit): Giá trị giới hạn số vòng lặp (thường được đặt cố định là 15 hoặc 16 tùy độ chính xác yêu cầu).
  - rs (1 bit): Tín hiệu đầu ra báo trạng thái (Result Status).
- Logic:
  - Phép so sánh số học: Module chuyển đổi các vector bit sang dạng số không dấu (unsigned) để thực hiện phép so sánh  $\geq$  hoặc  $\leq$ .
  - rs <= '1': Nếu số vòng lặp hiện tại i vẫn nằm trong phạm vi cho phép (nhỏ hơn hoặc bằng N), mạch xuất mức cao để thông báo hệ thống tiếp tục hoạt động.
  - rs <= '0': Khi i vượt quá giá trị N, tín hiệu sẽ chuyển xuống mức thấp. Đây là lệnh dừng để bộ điều khiển (FSM) biết rằng kết quả  $e^t$  đã sẵn sàng.

Bộ comparator0

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity comparator0 is
    port(
        z : in std_logic_vector(15 downto 0);
        cal : out std_logic
    );
end comparator0;

architecture dataflow of comparator0 is
begin
    cal <= '1' when (signed(z) >= 0) else '0';
end dataflow;

```

Đây là module bộ so sánh với số 0 để quyết định chiều xoay:

- Chức năng: Xác định dấu của giá trị Z hiện tại (đối số còn lại). Kết quả so sánh này là căn cứ duy nhất để hệ thống quyết định sẽ thực hiện phép Cộng hay Trừ ở các bước tiếp theo nhằm đưa Z dần về giá trị 0.

- Các cổng (Ports):
  - z (16 bit): Dữ liệu vào từ thanh ghi 16bit. Đây là giá trị đối số  $t$  đang được triệt tiêu dần qua từng vòng lặp.
  - cal (1 bit): Tín hiệu đầu ra điều khiển tính toán (Calculation Control).
- Logic:
  - signed(z): Chuyển đổi vector 16-bit sang kiểu số có dấu (signed). Điều này cực kỳ quan trọng vì nó cho phép mạch kiểm tra bit trọng số cao nhất (MSB) để biết số đó là dương hay âm theo hệ bù 2.
  - cal <= '1' khi z >= 0: Nếu Z mang giá trị dương hoặc bằng 0, tín hiệu cal sẽ ở mức cao ('1'). Lúc này, hệ thống sẽ điều khiển các bộ cộng/trừ thực hiện việc xoay theo chiều âm để giảm giá trị Z
  - else '0': Nếu Z nhỏ hơn 0, tín hiệu cal chuyển xuống mức thấp ('0'). Lúc này, hệ thống sẽ điều khiển xoay theo chiều ngược lại (chiều dương) để tăng Z về hướng 0.

#### 5.1.4. Bộ dịch phải

Bộ Shift Right 16 bit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity shift_right_signed is
  port(
    din : in signed(15 downto 0);
    sh : in std_logic_vector(4 downto 0);
    dout : out signed(15 downto 0)
  );
end entity;

architecture rtl of shift_right_signed is
begin
  process(din, sh)
  begin
    dout <= shift_right(din, to_integer(unsigned(sh)));
  end process;
end rtl;
```

Đây là module thực hiện phép dịch bit để thay thế phép nhân:

- Chức năng: Thực hiện phép nhân giá trị đầu vào với  $2^{-i}$  thông qua việc dịch chuyển các bit sang phải  $i$  lần. Đây là thành phần giúp thuật toán CORDIC không cần sử dụng bộ nhân (Multiplier) mà vẫn tính toán được các giá trị phức tạp, giúp tiết kiệm tài nguyên và tăng tốc độ xử lý của mạch.
- Các cổng (Ports):

- din (16 bit - signed): Dữ liệu vào là số có dấu. Đây là tọa độ  $X$  hoặc  $Y$  hiện tại cần được tỉ lệ xích.
- sh (5 bit): Tín hiệu điều khiển số lần dịch (Shift Amount). Giá trị này nhận từ chỉ số vòng lặp  $i$ .
- dout (16 bit - signed): Dữ liệu đầu ra sau khi đã được dịch.
- Logic:
  - Dịch đại số (Arithmetic Shift): Vì đầu vào là kiểu signed, hàm shift\_right của thư viện numeric\_std sẽ tự động thực hiện phép dịch đại số.
  - Bảo toàn bit dấu (Sign Extension): Khi các bit bị đẩy sang phải, thay vì thêm các số '0' vào bên trái, hàm này sẽ sao chép bit dấu (bit MSB) vào các vị trí trống. Điều này đảm bảo rằng số âm sau khi dịch vẫn giữ nguyên tính chất âm và giá trị đại số đúng (ví dụ:  $-4$  dịch phải 1-bit sẽ ra  $-2$ , thay vì ra một số dương cực lớn).
  - to\_integer(unsigned(sh)): Chuyển đổi mã nhị phân của số vòng lặp sang số nguyên để mạch hiểu được cần phải dịch bao nhiêu vị trí.

### 5.1.5. Look-Up Table – LUT

Bộ chuyển đổi địa chỉ

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity i_address is
  port(
    i_in  : in  std_logic_vector(4 downto 0);
    addr_out : out std_logic_vector(4 downto 0)
  );
end i_address;

architecture rtl of i_address is
begin
  addr_out <= std_logic_vector(unsigned(i_in) - 1);
end rtl;
```

Đây là module bộ tính toán địa chỉ để truy cập bảng tra cứu (LUT):

- Chức năng: Thực hiện việc chuyển đổi chỉ số vòng lặp hiện tại ( $i$ ) thành địa chỉ vật lý để truy xuất dữ liệu trong bộ lưu giá trị LUT. Vì trong toán học CORDIC Hyperbolic, các bước lặp thường được tính từ  $i = 1, 2, 3, \dots$ , nhưng bảng LUT luôn bắt đầu từ địa chỉ 0, nên khối này đóng vai trò cầu nối để lấy đúng hằng số tương ứng với từng bước.
- Các cổng (Ports):
  - i\_in (5 bit): Chỉ số vòng lặp hiện tại lấy từ đầu ra của thanh ghi reg5bit.
  - addr\_out (5 bit): Địa chỉ đầu ra dùng để kết nối trực tiếp vào chân địa chỉ của bộ nhớ cordic\_lut\_mem.

- Logic:

- Phép trừ số học (- 1): Module lấy giá trị vòng lặp hiện tại và trừ đi 1 đơn vị ( $Address = i - 1$ ).
- Ép kiểu dữ liệu:
  - unsigned(i\_in): Chuyển vector bit sang số không dấu để có thể thực hiện phép trừ.
  - std\_logic\_vector(...): Chuyển kết quả tính toán ngược lại dạng vector để đồng bộ với giao diện của các khối nhớ.

#### Bộ tăng chỉ số vòng lặp

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity inc1 is
    port (
        input: in std_logic_vector (4 downto 0);
        output: out std_logic_vector (4 downto 0)
    );
end inc1;
architecture dataflow of inc1 is
begin
    output <= input + 1;
end dataflow;
```

Đây là module thực hiện phép tăng giá trị để chuyển bước lặp:

- Chức năng: Thực hiện phép toán cộng thêm 1 đơn vị vào giá trị đầu vào. Trong hệ thống CORDIC, khối này dùng để tính toán chỉ số vòng lặp kế tiếp ( $i + 1$ ) từ chỉ số hiện tại, giúp mạch tiến dần qua các bước lặp từ 1 đến  $N$ .
- Các cổng (Ports):
  - input (5 bit): Nhận chỉ số vòng lặp hiện tại từ đầu ra của thanh ghi reg5bit.
  - output (5 bit): Xuất ra giá trị đã được tăng thêm 1. Giá trị này được đưa quay trở lại bộ hợp kênh (MUX) để chuẩn bị nạp vào thanh ghi cho lần tiếp theo.
- Logic:
  - output <= input + 1: Đây là một phép toán cộng số học đơn giản. Nhờ sử dụng thư viện std\_logic\_unsigned, trình biên dịch sẽ tự động hiểu đầu vào là một số không dấu và tổng hợp thành một mạch cộng (Adder) tối ưu trên phần cứng.
  - Vì được viết theo kiến trúc dataflow, giá trị đầu ra sẽ thay đổi ngay lập tức khi đầu vào thay đổi, đảm bảo chỉ số vòng lặp mới luôn sẵn sàng trước khi cạnh lên của xung clock tiếp theo đến.

#### +Bộ lưu trữ bảng Look-Up Table LUT

```
library ieee;
use ieee.std_logic_1164.all;
```

```

use ieee.numeric_std.all;

entity cordic_lut_mem is
    port (
        Addr    : in std_logic_vector(4 downto 0);
        Data_out : out std_logic_vector(15 downto 0)
    );
end entity;

architecture RTL of cordic_lut_mem is

    type rom_type is array(0 to 31) of std_logic_vector(15 downto 0);
    constant ROM : rom_type := (
        0 => x"1194",
        1 => x"082C",
        2 => x"0405",
        3 => x"0201",
        4 => x"0100",
        5 => x"0080",
        6 => x"0040",
        7 => x"0020",
        8 => x"0010",
        9 => x"0008",
        10 => x"0004",
        11 => x"0002",
        12 => x"0001",
        13 => x"0001",
        14 => x"0000",
        15 => x"0000",
        others => x"0000"
    );

begin
    Data_out <= ROM(to_integer(unsigned(Addr)));
end RTL;

```

Đây là module bộ nhớ tra cứu (Look-Up Table - LUT) lưu trữ các hằng số toán học:

- Chức năng: Lưu trữ các giá trị góc xoay Hyperbolic cố định  $\alpha_i = \operatorname{atanh}(2^{-i})$ . Thay vì tính toán các hàm toán học phức tạp này trực tiếp trên phần cứng (vốn cực kỳ tốn kém tài nguyên), các giá trị này được tính toán sẵn và lưu vào bộ nhớ ROM để truy xuất tức thời, giúp tối ưu hóa tốc độ xử lý của mạch.
- Các cổng (Ports):
  - Addr (5 bit): Chân địa chỉ đầu vào, nhận giá trị từ khối tính toán địa chỉ ( $i\_address$ ). Với 5 bit, bộ nhớ có thể chứa tối đa 32 hằng số.
  - Data\_out (16 bit): Dữ liệu đầu ra, cung cấp hằng số  $\operatorname{atanh}$  tương ứng với giá trị địa chỉ cho bộ cộng/trừ ở nhánh Z.
- Logic và Định dạng dữ liệu:
  - Định dạng số phẩy thập (Q3.13): Toàn bộ dữ liệu 16-bit được lưu trữ dưới định dạng Q3.13 (3 bit cho phần nguyên và 13 bit cho phần phân số).

- Ví dụ: Mã Hex x"1194" tại địa chỉ 0 tương ứng với giá trị thực  $\text{tê atanh}(2^{-1}) \approx 0.5493$ .
- Hằng số ROM (constant ROM): Đây là một mảng dữ liệu cố định không thay đổi. Các giá trị trong bảng giảm dần khi chỉ số  $i$  tăng lên, cho phép thuật toán thực hiện các bước xoay dần để tiến tới kết quả chính xác.
- Sử dụng câu lệnh gán đồng thời kết hợp hàm chuyển đổi kiểu dữ liệu `to_integer(unsigned(Addr))`. Điều này cho phép mạch lấy dữ liệu ra ngay lập tức ngay khi địa chỉ thay đổi (truy xuất không đồng bộ), giúp giảm độ trễ cho hệ thống.

### 5.1.6. Bộ tính toán

Bộ cộng

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity adder is
  port(
    A : in std_logic_vector(15 downto 0);
    B : in std_logic_vector(15 downto 0);
    SUM : out std_logic_vector(15 downto 0)
  );
end adder;

architecture dataflow of adder is
begin
  SUM <= std_logic_vector(signed(A) + signed(B));
end dataflow;
```

Đây là module thực hiện phép cộng hai số nhị phân 16-bit:

- Chức năng: Thực hiện phép cộng số học giữa hai giá trị đầu vào. Trong hệ thống CORDIC, các bộ cộng này được sử dụng song song để tính toán các giá trị cho bước lặp kế tiếp ( $X_i + \text{shift}_Y$ ,  $Y_i + \text{shift}_X$  hoặc  $Z_i + \text{LUT}$ ).
- Các cổng (Ports):
  - A, B (16 bit): Hai toán hạng đầu vào nhận dữ liệu từ các thanh ghi hoặc khối dịch bit.
  - SUM (16 bit): Kết quả của phép cộng  $A + B$ .
- Logic:
  - Phép gán đồng thời: Giúp kết quả SUM được cập nhật ngay lập tức khi A hoặc B thay đổi.
  - `signed(A)`, `signed(B)`: Ép kiểu dữ liệu đầu vào sang số có dấu. Điều này cực kỳ quan trọng để trình tổng hợp thiết kế mạch cộng bù 2 (Two's Complement), giúp xử lý chính xác các số âm trong quá trình tính toán tọa độ.

- `std_logic_vector(...)`: Sau khi cộng xong, kết quả được chuyển ngược lại dạng vector bit để tương thích với các cổng kết nối của các module khác trong hệ thống.

Bộ trừ

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity subtractor is
  port(
    A : in std_logic_vector(15 downto 0);
    B : in std_logic_vector(15 downto 0);
    SUB : out std_logic_vector(15 downto 0)
  );
end subtractor;

architecture dataflow of subtractor is
begin
  SUB <= std_logic_vector(signed(A) - signed(B));
end dataflow;
```

Đây là module thực hiện phép trừ hai số nhị phân 16-bit:

- Chức năng: Thực hiện phép trừ số học giữa hai giá trị đầu vào ( $A - B$ ). Trong hệ thống CORDIC, bộ trừ này hoạt động song song với bộ cộng để tính toán các giá trị tọa độ và góc xoay cho bước lặp kế tiếp khi điều kiện xoay yêu cầu giảm giá trị ( $X_i - \text{shift}_Y$ ,  $Y_i - \text{shift}_X$  hoặc  $Z_i - \text{LUT}$ ).
- Các cổng (Ports):
  - A (16 bit): Số bị trừ, thường nhận dữ liệu từ các thanh ghi lưu trữ hiện tại của X, Y, Z.
  - B (16 bit): Số trừ, nhận giá trị gia số từ khối dịch bit hoặc hằng số từ bảng LUT.
  - SUB (16 bit): Kết quả của phép tính hiệu  $A - B$ .
- Logic:
  - Phép gán đồng thời (Concurrent Assignment): Kết quả SUB được tính toán ngay lập tức thông qua mạch logic tổ hợp khi có sự thay đổi ở đầu vào A hoặc B, đảm bảo không gây trễ nhịp cho luồng dữ liệu.
  - Số học có dấu (Signed Arithmetic): Tương tự bộ cộng, việc ép kiểu `signed(A)` và `signed(B)` giúp mạch thực hiện phép trừ theo quy tắc số bù 2. Điều này cực kỳ quan trọng vì trong CORDIC, các giá trị tọa độ Y và góc Z thường xuyên mang giá trị âm.

#### 5.1.7. Datapath

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

```

entity datapath is
  port(
    clk    : in std_logic;
    rst    : in std_logic;
    x_sel  : in std_logic_vector(1 downto 0);
    y_sel  : in std_logic_vector(1 downto 0);
    z_sel  : in std_logic_vector(1 downto 0);
    i_sel  : in std_logic;
    x_en   : in std_logic;
    y_en   : in std_logic;
    z_en   : in std_logic;
    i_en   : in std_logic;
    i_in   : in std_logic_vector(4 downto 0);
    x_in   : in std_logic_vector(15 downto 0);
    y_in   : in std_logic_vector(15 downto 0);
    z_in   : in std_logic_vector(15 downto 0);
    cal    : out std_logic;
    rs     : out std_logic;
    exp_out : out std_logic_vector(15 downto 0)
  );
end datapath;

architecture structural of datapath is
  component cordic_lut_mem is
    port (
      Addr    : in std_logic_vector(4 downto 0);
      Data_out : out std_logic_vector(15 downto 0)
    );
  end component;

  component reg16bit is
    port(
      clk : in std_logic;
      D   : in std_logic_vector(15 downto 0);
      en  : in std_logic;
      rst : in std_logic;
      Q   : out std_logic_vector(15 downto 0)
    );
  end component;

  component reg5bit is
    port(
      clk : in std_logic;
      D   : in std_logic_vector(4 downto 0);
      en  : in std_logic;
      rst : in std_logic;
      Q   : out std_logic_vector(4 downto 0)
    );
  end component;

  component mux3to1 is
    port(
      sel : in std_logic_vector(1 downto 0);
      I0  : in std_logic_vector(15 downto 0);
      I1  : in std_logic_vector(15 downto 0);
      I2  : in std_logic_vector(15 downto 0);

```

```

        O : out std_logic_vector(15 downto 0)
    );
end component;

component mux2to1 is
    port(
        sel : in std_logic;
        I0 : in std_logic_vector(4 downto 0);
        I1 : in std_logic_vector(4 downto 0);
        O : out std_logic_vector(4 downto 0)
    );
end component;

component shift_right_signed is
    port(
        din : in signed(15 downto 0);
        sh : in std_logic_vector(4 downto 0);
        dout : out signed(15 downto 0)
    );
end component;

component comparatorN is
    port(
        i : in std_logic_vector(4 downto 0);
        N : in std_logic_vector(4 downto 0);
        rs : out std_logic
    );
end component;

component comparator0 is
    port(
        z : in std_logic_vector(15 downto 0);
        cal : out std_logic
    );
end component;

component inc1 is
    port (
        input : in std_logic_vector(4 downto 0);
        output : out std_logic_vector(4 downto 0)
    );
end component;

component i_address is
    port(
        i_in : in std_logic_vector(4 downto 0);
        addr_out : out std_logic_vector(4 downto 0)
    );
end component;

component adder is
    port(
        A : in std_logic_vector(15 downto 0);
        B : in std_logic_vector(15 downto 0);
        SUM : out std_logic_vector(15 downto 0)
    );
end component;

```

**end component;**

**component subtractor is**

**port**(

A : **in** std\_logic\_vector(15 downto 0);

B : **in** std\_logic\_vector(15 downto 0);

SUB : **out** std\_logic\_vector(15 downto 0)

);

**end component;**

**signal** x\_out\_mux, y\_out\_mux, z\_out\_mux : std\_logic\_vector(15 downto 0);

**signal** i\_out\_mux : std\_logic\_vector(4 downto 0);

**signal** x\_out, y\_out, z\_out : std\_logic\_vector(15 downto 0);

**signal** i\_out : std\_logic\_vector(4 downto 0);

**signal** x\_shift\_s, y\_shift\_s : signed(15 downto 0);

**signal** x1, y1, z1 : std\_logic\_vector(15 downto 0);

**signal** x2, y2, z2 : std\_logic\_vector(15 downto 0);

**signal** i\_inc : std\_logic\_vector(4 downto 0);

**signal** lut\_addr\_signal : std\_logic\_vector(4 downto 0);

**signal** s\_lut\_data : std\_logic\_vector(15 downto 0);

**begin**

U\_LUT: cordic\_lut\_mem

**port map**(

Addr => lut\_addr\_signal,

Data\_out => s\_lut\_data

);

MUX\_X: mux3to1 **port map**(sel => x\_sel, I0 => x\_in, I1 => x1, I2 => x2, O => x\_out\_mux);

MUX\_Y: mux3to1 **port map**(sel => y\_sel, I0 => y\_in, I1 => y1, I2 => y2, O => y\_out\_mux);

MUX\_Z: mux3to1 **port map**(sel => z\_sel, I0 => z\_in, I1 => z1, I2 => z2, O => z\_out\_mux);

MUX\_I: mux2to1 **port map**(sel => i\_sel, I0 => i\_in, I1 => i\_inc, O => i\_out\_mux);

REG\_X: reg16bit **port map**(clk => clk, D => x\_out\_mux, en => x\_en, rst => rst, Q => x\_out);

REG\_Y: reg16bit **port map**(clk => clk, D => y\_out\_mux, en => y\_en, rst => rst, Q => y\_out);

REG\_Z: reg16bit **port map**(clk => clk, D => z\_out\_mux, en => z\_en, rst => rst, Q => z\_out);

REG\_I: reg5bit **port map**(clk => clk, D => i\_out\_mux, en => i\_en, rst => rst, Q => i\_out);

INC\_I: inc1 **port map**(input => i\_out, output => i\_inc);

CALC\_I\_ADDR: i\_address

**port map**(i\_in => i\_out, addr\_out => lut\_addr\_signal);

SHIFT\_X: shift\_right\_signed **port map**(din => signed(x\_out), sh => i\_out, dout => x\_shift\_s);

SHIFT\_Y: shift\_right\_signed **port map**(din => signed(y\_out), sh => i\_out, dout => y\_shift\_s);

COMPARE\_Z: comparator0 **port map**(z => z\_out, cal => cal);

COMPARE\_I: comparator1N **port map**(i => i\_out, N => "01111", rs => rs);

ADD\_OUT\_X1: adder **port map**(A => x\_out, B => std\_logic\_vector(y\_shift\_s), SUM => x1);

ADD\_OUT\_Y1: adder **port map**(A => y\_out, B => std\_logic\_vector(x\_shift\_s), SUM => y1);

SUB\_OUT\_X2: subtractor **port map**(A => x\_out, B => std\_logic\_vector(y\_shift\_s), SUB => x2);

SUB\_OUT\_Y2: subtractor **port map**(A => y\_out, B => std\_logic\_vector(x\_shift\_s), SUB => y2);

SUB\_OUT\_Z1: subtractor **port map**(A => z\_out, B => s\_lut\_data, SUB => z1);

ADD\_OUT\_Z2: adder **port map**(A => z\_out, B => s\_lut\_data, SUM => z2);

ADD\_DONE: adder

```

port map(A => x_out, B => y_out, SUM => exp_out);
end structural;

```

Đây là module datapath, kết nối tất cả các linh kiện đơn lẻ thành một hệ thống hoàn chỉnh:

- Chức năng: Thực thi toàn bộ luồng tính toán của thuật toán CORDIC Hyperbolic. Nó kết nối các thành phần để biến đổi dữ liệu đầu vào thành kết quả hàm mũ  $e^t$ .
- Kiến trúc Structural (Cấu trúc):
  - Module không mô tả trực tiếp thuật toán mà sử dụng lệnh port map để "đi dây" nối các linh kiện đã thiết kế trước đó lại với nhau.
  - Sử dụng các tín hiệu nội bộ (signal) làm trung gian để kết nối dữ liệu giữa các khối.
- Luồng hoạt động chính:
  - Giai đoạn nạp: Các bộ MUX chọn dữ liệu ban đầu  $(1/K, 0, t)$  để nạp vào các thanh ghi  $X, Y, Z$ .
  - Giai đoạn lặp: Tại mỗi chu kỳ, dữ liệu từ thanh ghi được đưa đi dịch bit và tra cứu bảng LUT. Các bộ cộng/trừ sẽ tính toán sẵn các giá trị mới.
  - Giai đoạn điều khiển: Các bộ so sánh gửi kết quả đầu ra đến Control Unit làm đầu vào, quyết định các trạng thái và tín hiệu điều khiển trong Controller.
  - Giai đoạn kết quả: Khi quá trình lặp dừng lại, bộ cộng cuối cùng ADD\_DONE thực hiện phép tính  $X + Y$  để xuất ra giá trị  $e^t$  tại cổng exp\_out.

## 5.2. Control Unit :

### 5.2.1. IIR và OIR

#### IIR:

Đây là module tạo thanh ghi lưu giá trị của mảng đầu vào trong Main Memory

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity IIR is
  port(
    clk  : in std_logic;
    rst  : in std_logic;
    IIRinc : in std_logic;
    Q    : out std_logic_vector(5 downto 0)
  );
end IIR;

architecture behavioral of IIR is

```

```

signal IIR_temp : unsigned(5 downto 0);
begin
  process(clk, rst)
  begin
    if rst = '1' then
      IIR_temp <= (others => '0');
    elsif rising_edge(clk) then
      if IIRinc = '1' then
        if IIR_temp = 31 then
          IIR_temp <= (others => '0');
        else
          IIR_temp <= IIR_temp + 1;
        end if;
      end if;
    end if;
  end process;
  Q <= std_logic_vector(IIR_temp);
end behavioral;

```

Các thành phần trong module;

- Entity: Khai báo giao diện đầu vào ra
  - rst, clk: Tín hiệu reset và clock chung của mạch
  - IIRinc: Tín hiệu để tăng thanh ghi lên 1
  - Q: Đầu ra của thanh ghi
- Achitechture: Mô tả hoạt động:
  - Khai báo 1 tín hiệu IIR\_temp để lưu giá trị thanh ghi
  - Đầu tiên là kiểm tra tín hiệu reset, nếu rst='1' thì sẽ đặt lại IIR\_temp về vị trí ban đầu là 0.
  - Các tín hiệu cập nhật sẽ xảy ra khi có xung clock, nếu tín hiệu IIRinc='1' thì sẽ tăng tín hiệu IRR\_temp lên 1, tối đa tăng đến 31 và quay lại, ngược lại không có gì xảy ra
  - Gán đầu ra O bằng IRR\_temp để đẩy output ra ngoài.

## OIR:

Đây là module tạo thanh ghi lưu giá trị của mảng đầu ra trong Main Memory

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity OIR is
  port(
    clk  : in std_logic;
    rst  : in std_logic;
    OIRinc : in std_logic;
    Q    : out std_logic_vector(5 downto 0)
  );
end OIR;

```

```

architecture behavioral of OIR is
  constant OFFSET : unsigned(5 downto 0) := "100000";
  signal OIR_temp : unsigned(5 downto 0);
begin
  process(clk, rst)
  begin
    if rst = '1' then
      OIR_temp <= OFFSET;
    elsif rising_edge(clk) then
      if OIRinc = '1' then
        if OIR_temp = 63 then
          OIR_temp <= OFFSET;
        else
          OIR_temp <= OIR_temp + 1;
        end if;
      end if;
    end if;
  end process;
  Q <= std_logic_vector(OIR_temp);
end behavioral;

```

Module này hoạt động tương tự IIR, nó sẽ xuất phát ở địa chỉ thứ 32 của mem và kết thúc ở địa chỉ 63. Các thành phần trong module:

- Entity: Khai báo giao diện đầu vào ra
  - rst, clk: Tín hiệu reset và clock chung của mạch
  - OIRinc: Tín hiệu để tăng thanh ghi lên 1
  - Q: Đầu ra của thanh ghi
- Achitechture: Mô tả hoạt động:
  - Khai báo 1 tín hiệu OIR\_temp để lưu giá trị thanh ghi, khai báo OFFSET là 10000(=32) để lưu điểm bắt đầu
  - Đầu tiên là kiểm tra tín hiệu reset, nếu rst='1' thì sẽ đặt lại OIR\_temp về vị trí ban đầu là OFFSET.
  - Các tín hiệu cập nhật sẽ xảy ra khi có xung clock, nếu tín hiệu OIRinc='1' thì sẽ tăng tín hiệu ORR\_temp lên 1, tối đa tăng đến 63 và quay lại, ngược lại không có gì xảy ra
  - Gán đầu ra O bằng OIR\_temp để đẩy output ra ngoài.

### 5.2.2. Memory – Bộ nhớ vào ra

Bộ nhớ này chứa input tính toán đầu vào và output đầu ra, tác dụng như 1 buffer để CPU có thể gán nhiều phép tính cho Exponential

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

```

```

entity memory is
  port(
    clk    : in std_logic;
    rst    : in std_logic;
    Mre    : in std_logic;
    Mwe    : in std_logic;
    address : in std_logic_vector(5 downto 0);
    data_in : in std_logic_vector(15 downto 0);
    data_out : out std_logic_vector(15 downto 0)
  );
end entity;

architecture behav of memory is
  type mem_type is array (0 to 63) of std_logic_vector(15 downto 0);
  signal tmp_mem : mem_type;
begin
  process(clk, rst)
  begin
    if rst = '1' then
      tmp_mem(0) <= x"8000"; -- -4.00
      tmp_mem(1) <= x"9000"; -- -3.50
      tmp_mem(2) <= x"A000"; -- -3.00
      tmp_mem(3) <= x"B000"; -- -2.50
      tmp_mem(4) <= x"C000"; -- -2.00
      tmp_mem(5) <= x"D000"; -- -1.50
      tmp_mem(6) <= x"E000"; -- -1.00
      tmp_mem(7) <= x"E800"; -- -0.75
      tmp_mem(8) <= x"F000"; -- -0.50
      tmp_mem(9) <= x"F800"; -- -0.25
      tmp_mem(10) <= x"0000"; -- 0.00
      tmp_mem(11) <= x"0800"; -- +0.25
      tmp_mem(12) <= x"1000"; -- +0.50
      tmp_mem(13) <= x"1800"; -- +0.75
      tmp_mem(14) <= x"2000"; -- +1.00
      tmp_mem(15) <= x"2400"; -- +1.125
      tmp_mem(16) <= x"2800"; -- +1.250
      tmp_mem(17) <= x"2B33"; -- +1.350
      tmp_mem(18) <= x"2C28"; -- +1.380

      for i in 19 to 63 loop
        tmp_mem(i) <= (others => '0');
      end loop;

      data_out <= (others => 'Z');

    elsif rising_edge(clk) then
      if Mwe = '1' then
        tmp_mem(to_integer(unsigned(address))) <= data_in;
      end if;

      if Mre = '1' then
        data_out <= tmp_mem(to_integer(unsigned(address)));
      else
        data_out <= (others => 'Z');
      end if;
    end if;
  end process;
end architecture;

```

```

end if;
end process;
end behav;

```

Các thành phần trong modules:

- Entity: Khai báo giao diện đầu vào ra
  - rst, clk: Tín hiệu reset và clock chung của mạch
  - Mre, Mwe: Tín hiệu cho phép đọc ghi xảy ra trong memory
  - address: Tín hiệu địa chỉ xảy ra đọc ghi
  - data\_in, data\_out: đầu vào, ra tương ứng với Mre, Mwe
- Achitecture: Mô tả hoạt động:
  - Khai báo 1 kiểu dữ liệu mem\_type là mảng signal chứa được 64 signal 16-bits.
  - Khai báo mảng tmp\_mem thuộc kiểu dữ liệu trên.
  - Khi có tín hiệu reset, memory sẽ được set hết về 0, tuy nhiên bài này không mô phỏng cách CPU truyền dữ liệu vào mảng nên 19 giá trị đầu ta đặt sẵn các giá trị input để mô phỏng cách mạch hoạt động
  - Khi không có Mre hoặc Mwe hoặc reset, mạch sẽ không chiếm đường truyền, nên giá trị xuất ra là ZZZZ.
  - Khi có xung clock: ưu tiên Mwe hơn Mre
    - Nếu Mwe='1', giá trị tại address hiện tại được ghi từ data\_in.
    - Nếu Mre='1', giá trị tại address hiện tại được ghi ra data\_out.

### 5.2.3. *Controller – Module điều khiển*

Đầu tiên ta khai báo giao diện vào ra cho module:

```

entity controller is
port (
  start : in std_logic;
  rst   : in std_logic;
  clk   : in std_logic;
  rs    : in std_logic;
  cal   : in std_logic;

  x_sel : out std_logic_vector(1 downto 0);
  y_sel : out std_logic_vector(1 downto 0);
  z_sel : out std_logic_vector(1 downto 0);
  i_sel : out std_logic;

  x_en : out std_logic;
  y_en : out std_logic;
  z_en : out std_logic;
  i_en : out std_logic;

  Mre : out std_logic;
  Mwe : out std_logic;
  addr_sel: out std_logic;
  IIRinc : out std_logic;

```

```

OIRinc : out std_logic;

done  : out std_logic
);
end controller;

```

Như đã biết, controller là 1 máy trạng thái, vì vậy ta khai báo các trạng thái từ S0->S9 trong một process:

```

architecture fsm of controller is
  type state_type is (S0, S1, S2, S3, S4, S5, S6, S7, S8, S9);
  signal state : state_type;
begin
  process(clk, rst)
  begin
    if rst = '1' then
      state <= S0;
    elsif rising_edge(clk) then
      case state is
        when S0 =>
          state <= S1;

        when S1 =>
          if start = '1' then
            state <= S2;
          else
            state <= S1;
          end if;

          when S2 =>
            state <= S3;

        when S3 =>
          state <= S4;

        when S4 =>
          if rs = '1' then
            state <= S5;
          else
            state <= S8;
          end if;

        when S5 =>
          if cal = '1' then
            state <= S6;
          else
            state <= S7;
          end if;

        when S6 =>
          state <= S4;

```

```

when S7 =>
    state <= S4;

when S8 =>
    state <= S9;

        when S9 =>
            if start = '0' then
                state <= S0;
            else
                state <= S9;
            end if;

when others =>
    state <= S0;
end case;
end if;
end process;

```

Cũng trong architecture của module, ta gán các tín hiệu tương ứng với các trạng thái mà ta đã giải thích trong phần FSM bên trên:

```

x_sel <= "00" when state = S3 else
    "01" when state = S6 else
    "10" when state = S7 else
    "00";

y_sel <= "00" when state = S3 else
    "01" when state = S6 else
    "10" when state = S7 else
    "00";

z_sel <= "00" when state = S3 else
    "01" when state = S6 else
    "10" when state = S7 else
    "00";

i_sel <= '0' when state = S3 else '1' when state = S6 or state = S7;

x_en <= '1' when state = S3 or state = S6 or state = S7 else '0';
y_en <= '1' when state = S3 or state = S6 or state = S7 else '0';
z_en <= '1' when state = S3 or state = S6 or state = S7 else '0';
i_en <= '1' when state = S3 or state = S6 or state = S7 else '0';

Mre <= '1' when state = S2 else '0';
addr_sel <= '1' when state = S8 else '0';
IIRinc <= '1' when state = S3 else '0';
Mwe <= '1' when state = S8 else '0';
OIRinc <= '1' when state = S8 else '0';
done <= '1' when state = S8 else '0';

```

```
end fsm;
```

Như vậy Module controller sẽ mô tả máy trạng thái của đơn vị điều khiển qua từng trạng thái và trong mỗi trạng thái ta cần các tín hiệu gì và chúng được gán giá trị bao nhiêu.

#### 5.2.4. *Control\_Unit: Kết hợp controller với các biến điều khiển địa chỉ Memory*

Control\_Unit được viết ra bằng cách tạo đường nối dữ liệu giữa Controller và OIR, IIR, cũng như đường dữ liệu từ bên ngoài vào. Các bước để viết:

- Khai báo entity: Các đầu vào ra để giao tiếp với Control\_Unit

```
entity control_unit is
  port (
    clk      : in std_logic;
    rst      : in std_logic;
    start    : in std_logic;
    rs       : in std_logic;
    cal      : in std_logic;
    x_sel    : out std_logic_vector(1 downto 0);
    y_sel    : out std_logic_vector(1 downto 0);
    z_sel    : out std_logic_vector(1 downto 0);
    i_sel    : out std_logic;
    x_en     : out std_logic;
    y_en     : out std_logic;
    z_en     : out std_logic;
    i_en     : out std_logic;
    Mre      : out std_logic;
    Mwe      : out std_logic;
    address  : out std_logic_vector(5 downto 0);
    done     : out std_logic
  );
end control_unit;
```

- Khai báo component cho Controller và IIR, OIR và muxAddress để có thể sử dụng chúng trong code

```
architecture structural of control_unit is

  component controller is
    port (
      start, rst, clk, rs, cal : in std_logic;
      x_sel, y_sel, z_sel : out std_logic_vector(1 downto 0);
      i_sel, x_en, y_en, z_en, i_en : out std_logic;
      Mre, Mwe, addr_sel, IIRinc, OIRinc : out std_logic;
      done : out std_logic
    );
  end component;
```

```

component IIR is
  port(
    clk, rst, IIRinc : in std_logic;
    Q : out std_logic_vector(5 downto 0)
  );
end component;

component OIR is
  port(
    clk, rst, OIRinc : in std_logic;
    Q : out std_logic_vector(5 downto 0)
  );
end component;

component muxAddress is
  port(
    sel : in std_logic;
    I0, I1 : in std_logic_vector(5 downto 0);
    O : out std_logic_vector(5 downto 0)
  );
end component;

```

- Khai báo các tín hiệu trung gian để kết nối:

```

signal s_addr_sel : std_logic;
signal s_IIRinc : std_logic;
signal s_OIRinc : std_logic;
signal s_IIR_out : std_logic_vector(5 downto 0);
signal s_OIR_out : std_logic_vector(5 downto 0);

```

- Cuối cùng, kết nối các tín hiệu vào giữa Controller và các tín hiệu liên quan

```

begin

  U_CTRL: controller
    port map (
      clk    => clk,
      rst    => rst,
      start  => start,
      rs     => rs,
      cal    => cal,
      x_sel  => x_sel,
      y_sel  => y_sel,
      z_sel  => z_sel,
      i_sel  => i_sel,
      x_en   => x_en,
      y_en   => y_en,
      z_en   => z_en,
      i_en   => i_en,

```

```

    Mre    => Mre,
    Mwe    => Mwe,
    addr_sel => s_addr_sel,
    IIRinc  => s_IIRinc,
    OIRinc  => s_OIRinc,
    done    => done
);

U_IIR: IIR
  port map (
    clk    => clk,
    rst    => rst,
    IIRinc => s_IIRinc,
    Q      => s_IIR_out
  );

U_OIR: OIR
  port map (
    clk    => clk,
    rst    => rst,
    OIRinc => s_OIRinc,
    Q      => s_OIR_out
  );

U_MUX_ADDR: muxAddress
  port map (
    sel => s_addr_sel,
    I0  => s_IIR_out,
    I1  => s_OIR_out,
    O    => address
  );

end structural;

```

### 5.2.5. Microprocessor: Vì xử lý cho việc tính Exponential

Microprocessor được viết ra bằng cách tạo đường nối dữ liệu giữa Datapath và Control\_Unit, cũng như đường dữ liệu từ bên ngoài vào. Các bước để viết:

- Khai báo entity: Các đầu vào ra để giao tiếp với microprocessor

```

entity microprocessor is
  port (
    clk      : in std_logic;
    rst      : in std_logic;
    start     : in std_logic;

    mem_data_in : in std_logic_vector(15 downto 0);
    mem_addr    : out std_logic_vector(5 downto 0);
    mem_data_out: out std_logic_vector(15 downto 0);
    Mre         : out std_logic;
    Mwe         : out std_logic;

```

```

    done      : out std_logic
);
end microprocessor;

```

- Khai báo component cho Datapath và Control\_Unit để có thể sử dụng chúng trong code

```

architecture structural of microprocessor is

    component control_unit is
        port (
            clk      : in std_logic;
            rst      : in std_logic;
            start    : in std_logic;
            rs       : in std_logic;
            cal      : in std_logic;

            x_sel    : out std_logic_vector(1 downto 0);
            y_sel    : out std_logic_vector(1 downto 0);
            z_sel    : out std_logic_vector(1 downto 0);
            i_sel    : out std_logic;
            x_en     : out std_logic;
            y_en     : out std_logic;
            z_en     : out std_logic;
            i_en     : out std_logic;

            Mre      : out std_logic;
            Mwe      : out std_logic;
            address  : out std_logic_vector(5 downto 0);
            done     : out std_logic
        );
    end component;

    component datapath is
        port(
            clk      : in std_logic;
            rst      : in std_logic;
            x_sel    : in std_logic_vector(1 downto 0);
            y_sel    : in std_logic_vector(1 downto 0);
            z_sel    : in std_logic_vector(1 downto 0);
            i_sel    : in std_logic;
            x_en     : in std_logic;
            y_en     : in std_logic;
            z_en     : in std_logic;
            i_en     : in std_logic;
            i_in     : in std_logic_vector(4 downto 0);
            x_in     : in std_logic_vector(15 downto 0);
            y_in     : in std_logic_vector(15 downto 0);
            z_in     : in std_logic_vector(15 downto 0);
            cal      : out std_logic;
            rs       : out std_logic;
            exp_out  : out std_logic_vector(15 downto 0)
        );

```

**end component;**

- Khai báo các tín hiệu trung gian để kết nối:

```
signal s_x_sel, s_y_sel, s_z_sel : std_logic_vector(1 downto 0);
signal s_i_sel      : std_logic;
signal s_x_en, s_y_en, s_z_en  : std_logic;
signal s_i_en      : std_logic;
signal s_rs, s_cal      : std_logic;

constant X_INIT_VAL : std_logic_vector(15 downto 0) := x"2690";
constant Y_INIT_VAL : std_logic_vector(15 downto 0) := x"0000";
constant I_START    : std_logic_vector(4 downto 0) := "00001";
```

- Cuối cùng, kết nối các tín hiệu vào giữa Datapath và Control\_Unit

```
begin

U_CU: control_unit
  port map (
    clk  => clk,
    rst  => rst,
    start => start,
    rs   => s_rs,
    cal  => s_cal,

    x_sel => s_x_sel,
    y_sel => s_y_sel,
    z_sel => s_z_sel,
    i_sel => s_i_sel,
    x_en  => s_x_en,
    y_en  => s_y_en,
    z_en  => s_z_en,
    i_en  => s_i_en,

    Mre  => Mre,
    Mwe  => Mwe,
    address => mem_addr,
    done  => done
  );

U_DP: datapath
  port map (
    clk  => clk,
    rst  => rst,

    x_sel => s_x_sel,
    y_sel => s_y_sel,
    z_sel => s_z_sel,
    i_sel => s_i_sel,
```

```

x_en => s_x_en,
y_en => s_y_en,
z_en => s_z_en,
i_en => s_i_en,

i_in => I_START,
x_in => X_INIT_VAL,
y_in => Y_INIT_VAL,
z_in => mem_data_in,

cal => s_cal,
rs => s_rs,
exp_out => mem_data_out
);

```

**end structural;**

### 5.3. Microprocessor + Memory = Top System :

Top system là module để liên kết Microprocessor với Memory và các đường dữ liệu ngoài, ta cũng viết nó như Microprocessor, gồm các bước:

- Khai báo entity:

```

entity cordic_system_top is
  port (
    clk    : in std_logic;
    rst    : in std_logic;
    start  : in std_logic;
    done   : out std_logic
  );
end entity;

```

- Khai báo component:

```

architecture structural of cordic_system_top is

  component microprocessor is
    port (
      clk      : in std_logic;
      rst      : in std_logic;
      start    : in std_logic;
      mem_data_in : in std_logic_vector(15 downto 0);
      mem_addr   : out std_logic_vector(5 downto 0);
      mem_data_out : out std_logic_vector(15 downto 0);
      Mre        : out std_logic;
      Mwe        : out std_logic;
    );
  end component;

```

```

        done      : out std_logic
    );
end component;

component memory is
    port (
        clk      : in  std_logic;
        rst      : in  std_logic;
        Mre      : in  std_logic;
        Mwe      : in  std_logic;
        address   : in  std_logic_vector(5 downto 0);
        data_in   : in  std_logic_vector(15 downto 0);
        data_out  : out std_logic_vector(15 downto 0)
    );
end component;

```

- Khai báo các tín hiệu trung gian để kết nối:

```

signal s_mem_addr  : std_logic_vector(5 downto 0);
signal s_data_p_to_m : std_logic_vector(15 downto 0);
signal s_data_m_to_p : std_logic_vector(15 downto 0);
signal s_mre, s_mwe : std_logic;

```

- Kết nối các tín hiệu để tạo thành top\_system

```

begin

U_PROCESSOR: microprocessor
    port map (
        clk      => clk,
        rst      => rst,
        start    => start,

        mem_data_in => s_data_m_to_p,
        mem_addr   => s_mem_addr,
        mem_data_out => s_data_p_to_m,

        Mre       => s_mre,
        Mwe       => s_mwe,

        done      => done
    );

U_MEMORY: memory
    port map (
        clk      => clk,
        rst      => rst,

        Mre      => s_mre,
        Mwe      => s_mwe,

```

```

        address => s_mem_addr,
        data_in => s_data_p_to_m,
        data_out => s_data_m_to_p
    );

```

```

end structural;

```

## 6. Mô phỏng/thực thi và đánh giá: Testbench:

Để kiểm tra kết quả và đưa ra đánh giá chi tiết, ta sử dụng phần mềm ModelSim để mô phỏng và xem dạng sóng chi tiết.

Testbench được viết như sau:

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity cordic_system_top_tb is
end entity;

architecture sim of cordic_system_top_tb is

    -- 1. Khai báo Component Top Level
    component cordic_system_top is
        port (
            clk    : in std_logic;
            rst    : in std_logic;
            start   : in std_logic;
            done    : out std_logic
        );
    end component;

    signal clk    : std_logic := '0';
    signal rst    : std_logic := '0';
    signal start  : std_logic := '0';
    signal done   : std_logic;

    constant CLK_PERIOD : time := 10 ns;

begin
    DUT: cordic_system_top
        port map (
            clk => clk,
            rst => rst,
            start => start,
            done => done
        );
    clk_process : process
    begin
        while now < 10000 ns loop
            clk <= '0';

```

```

    wait for CLK_PERIOD/2;
    clk <= '1';
    wait for CLK_PERIOD/2;
end loop;
wait;
end process;

stimulus_process : process
begin
    report "Starting Simulation...";
    start <= '0';
    rst <= '1';
    wait for 30 ns;
    rst <= '0';
    wait for 20 ns;

    for i in 0 to 18 loop
        report "Processing sample index: " & integer'image(i);

        wait until rising_edge(clk);
        start <= '1';

        wait until done = '1';
        start <= '0';
        wait for 2 * CLK_PERIOD;

        report "Sample " & integer'image(i) & " completed.";
    end loop;

    report "SUCCESS: All samples have been processed and stored in RAM.";

    wait for 100 ns;
    assert false report "End of Simulation" severity failure;
    wait;
end process;
end sim;

```

Đây là module mô phỏng (Testbench) dùng để kiểm tra tính đúng đắn của toàn bộ hệ thống CORDIC:

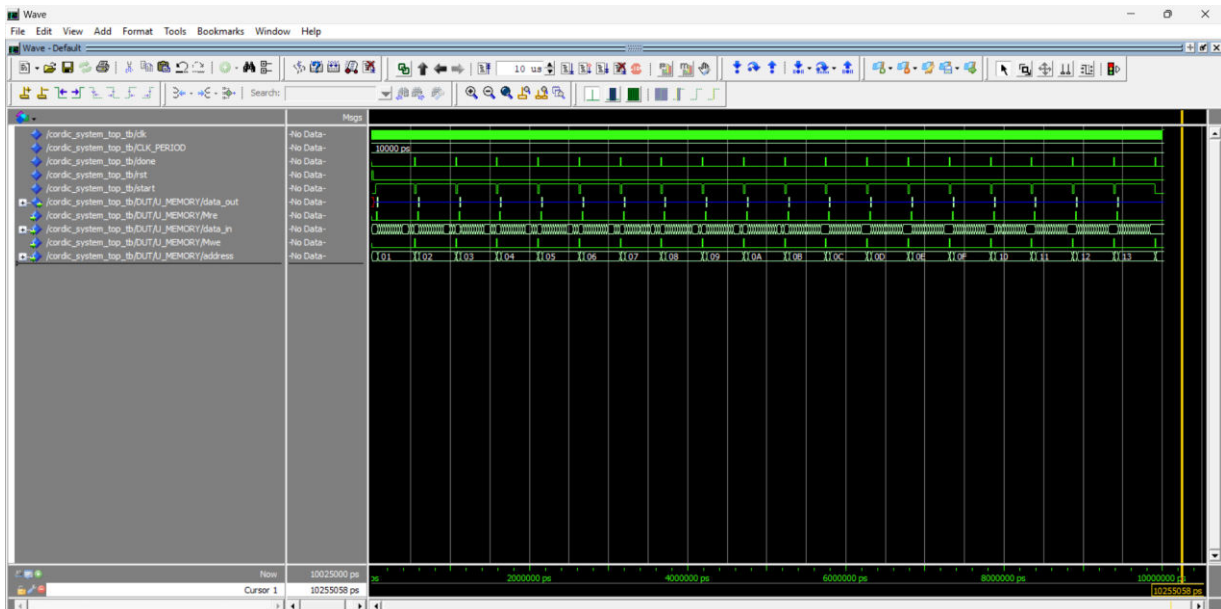
- Chức năng: Tạo ra các điều kiện giả lập (xung clock, tín hiệu reset, lệnh start) để vận hành hệ thống cordic\_system\_top trong môi trường mô phỏng. Kiểm tra xem hệ thống có xử lý đúng 19 mẫu dữ liệu và lưu vào RAM thành công hay không.
- Các thành phần chính:
  - DUT (Device Under Test): Thành phần đang được kiểm tra là cordic\_system\_top.
  - Tín hiệu mô phỏng: Các tín hiệu clk, rst, start được khởi tạo giá trị ban đầu để bắt đầu quá trình mô phỏng.
  - Hằng số CLK\_PERIOD: Định nghĩa chu kỳ xung nhịp là 10 ns (tương đương tần số 100 MHz).

- Các tiến trình xử lý (Processes):

- clk\_process (Tạo xung nhịp): Sử dụng một vòng lặp để tạo ra các cạnh lên và cạnh xuống của xung clock liên tục. Tiến trình này được thiết lập chạy trong 10.000 ns để đảm bảo đủ thời gian cho toàn bộ quá trình tính toán.
- stimulus\_process (Tạo kích bản kiểm thử):
  1. Khởi tạo: Gửi báo cáo "Starting Simulation", sau đó kích hoạt tín hiệu rst (Reset) trong 30 ns để đưa hệ thống về trạng thái ban đầu ổn định.
  2. Vòng lặp xử lý (0 to 18): Testbench thực hiện lặp 19 lần. Tại mỗi lần lặp, nó bật tín hiệu start để yêu cầu hệ thống tính toán một mẫu dữ liệu mới.
  3. Testbench sẽ "đợi" cho đến khi tín hiệu done từ hệ thống lên mức cao (báo hiệu tính toán và lưu RAM xong) thì mới hạ tín hiệu start xuống và chuẩn bị cho giá trị tiếp theo.
  4. Kết thúc: Sau khi hoàn thành tất cả mẫu dữ liệu, nó xuất thông báo "SUCCESS" và sử dụng lệnh assert false để dừng trình mô phỏng.

Kết quả mô phỏng:

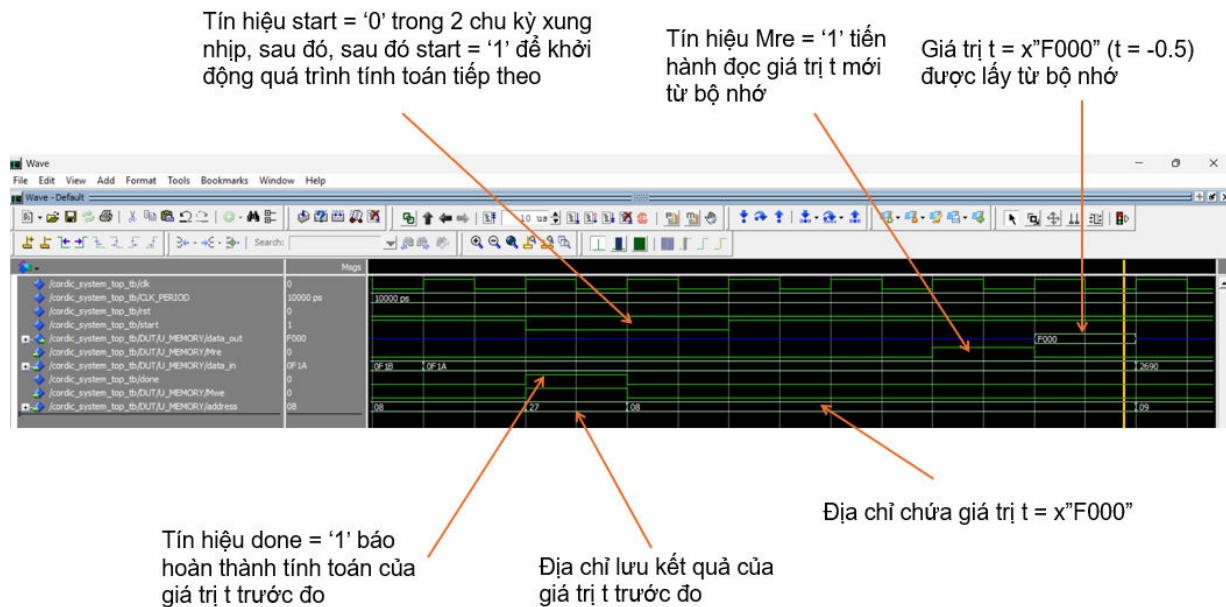
-Hình ảnh mô phỏng tổng thể: Có tất cả 19 giá trị t cần mô phỏng nên ta thấy địa chỉ chạy từ 00 -> 12 (tương đương 0 – 18 trong hệ 10) và 19 tín hiệu done chứng tỏ đã hoàn thành việc tính toán tất cả các giá trị. Để kiểm tra sự chính xác, ta sẽ đi kiểm tra chi tiết các giá trị.



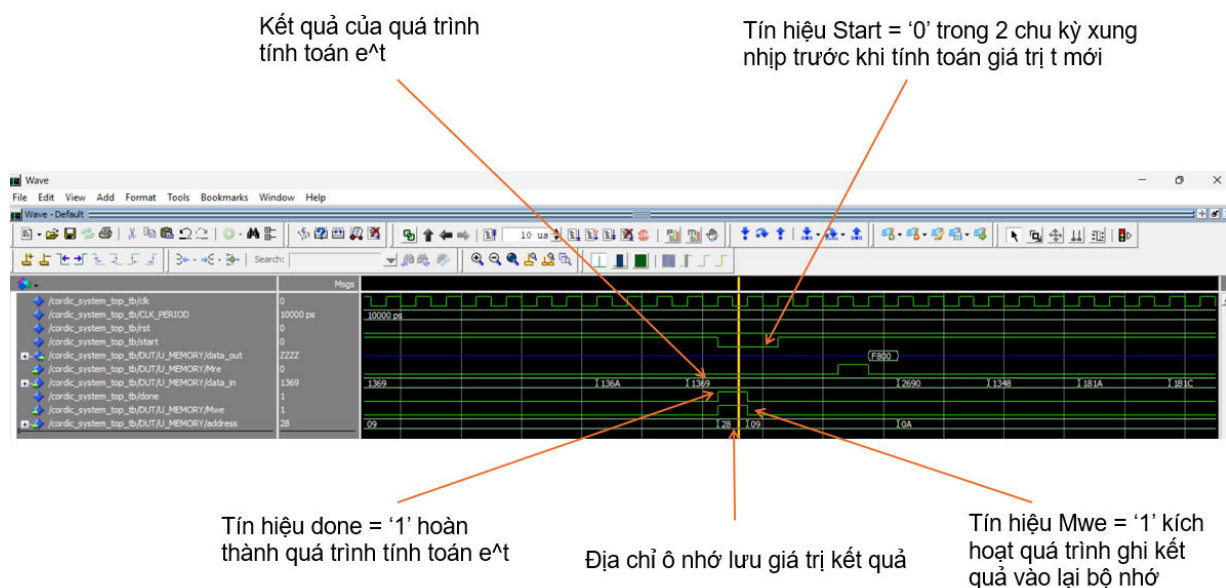
-Do trong Memory lưu tất cả là 19 giá trị t nằm ở địa chỉ từ 0 đến 18, nên để kiểm tra tính đúng của dữ liệu mô phỏng ta sẽ xem xét chi tiết một vài giá trị để thấy được kết quả chi tiết nhất. Ta sẽ quan sát với 3 giá trị  $t = -0.5$ ,  $t = 0$  và  $t = 0.5$ .

+Với  $t = -0.5$  (hệ 10) tương đương x"F000" (Q3.13 – hệ 16)

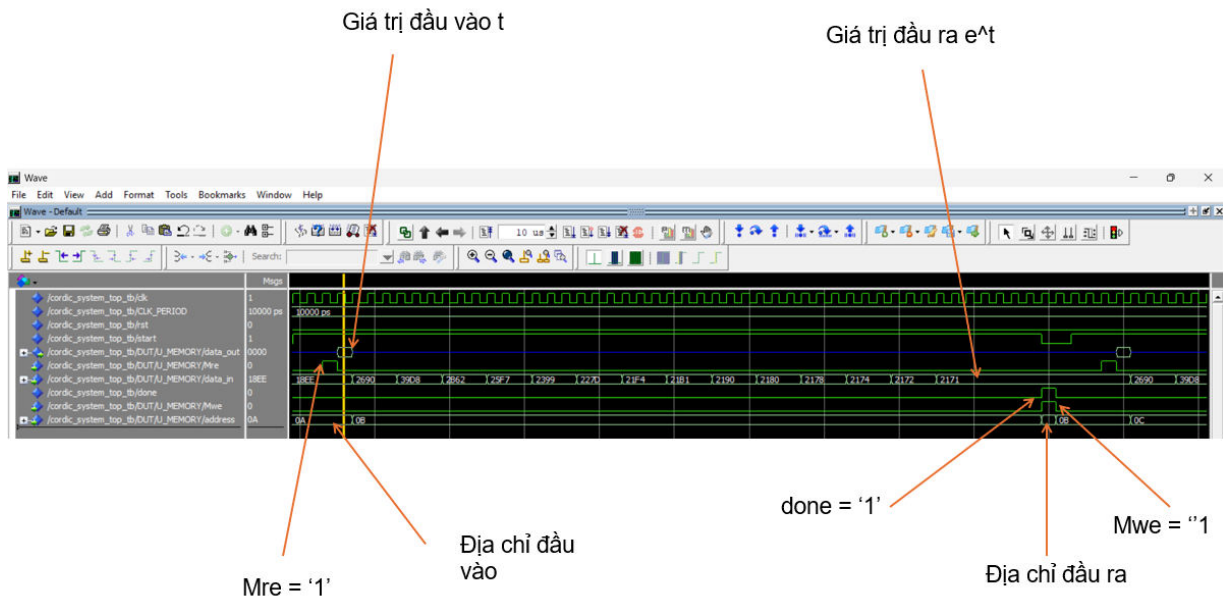
Khi hoàn thành quá trình tính toán với giá trị  $t$  trước đó (tức là  $t = -0.75$ ) tín hiệu start = '0' trong 2 chu kỳ xung nhịp, sau đó bắt đầu vòng lặp tiếp theo, tín hiệu start = '1' để khởi động quá trình tính toán. Đầu tiên, tín hiệu Mre = '1' kích hoạt quá trình đọc giá trị  $t$  từ Memory, sau đó Mre = '0' kết thúc quá trình đọc, lúc này data\_out tức giá trị  $t$  hay  $z\_init$  sẽ nhận giá trị  $t$  mới là "F000".



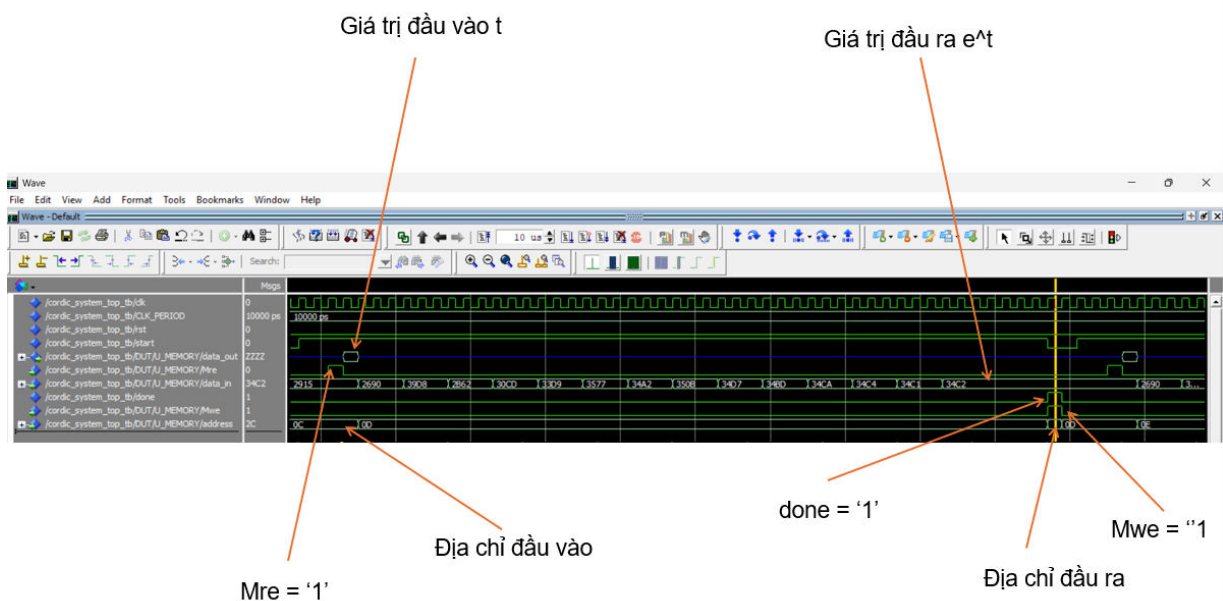
Tiếp theo việc tính toán bắt đầu diễn ra cho đến khi done = '1'. Ngay khi done = '1', tín hiệu Mwe cũng đồng thời = '1' để tiến hành quá trình ghi kết quả  $e^t$  (ở đây là  $x"1369"$  - Q3.13 hệ 16) vào bộ nhớ có địa chỉ  $x"28"$  (ô nhớ thứ 40 – hệ 10).



+Với  $t = 0$  (hệ 10) tương đương  $x''0000$  (Q3.13 hệ 16). Quá trình diễn ra tương tự so với  $t = -0.5$ . Ta sẽ quan sát hình ảnh mô phỏng.



+Với  $t = 0.5$  (hệ 10) tương đương  $x''1000$  (Q3.13 hệ 16). Quá trình diễn ra tương tự so với  $t = -0.5$ . Ta sẽ quan sát hình ảnh mô phỏng.



Phân tích kết quả mô phỏng:

Ta có bảng so sánh giá trị kết quả mô phỏng và kết quả thực tế:

### Giới hạn của định dạng Q3.13

Định dạng Q3.13 có 3-bit phần nguyên (bao gồm 1-bit dấu) và 13-bit phần phân số.

Giá trị lớn nhất (Max):

$$2^{(3-1)} - 2^{-13} = 4 - 0.000122 = 3.9998779$$

Giá trị nhỏ nhất (Min):

$$-2^{(3-1)} = -4.0$$

Việc lựa chọn dải giá trị thử nghiệm cho biến  $t$  từ  $-4$  đến  $1.38$  được tính toán dựa trên giới hạn lưu trữ của định dạng số cố định Q3.13. Do định dạng Q3.13 có giá trị biểu diễn tối đa là  $3.9998$ , nên để tránh hiện tượng tràn số (overflow) cho kết quả đầu ra  $e^t$ , biến  $t$  phải thỏa mãn điều kiện

$t \leq \ln(3.9998) \approx 1.386$ . Trong bảng mô phỏng, giá trị  $t = 1.38$  được chọn làm điểm cực đại để kiểm tra độ chính xác của hệ thống tại sát ngưỡng tràn số.

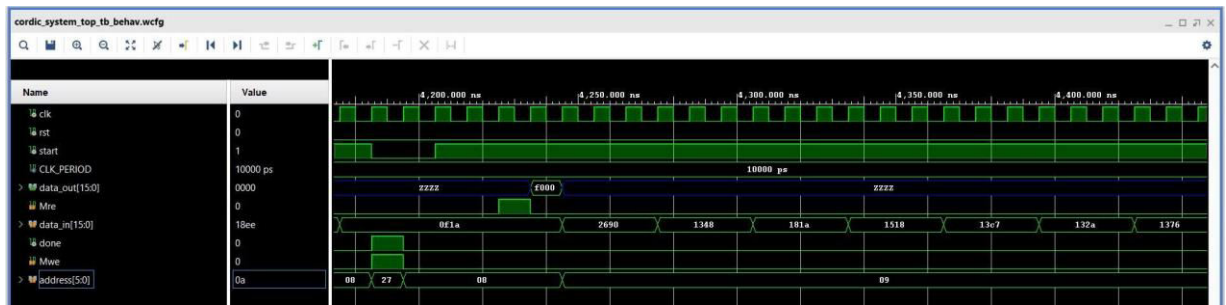
Bảng so sánh giá trị mô phỏng so với giá trị thực tế									
STT	Địa chỉ IN (Hệ 16)	Input t (Thập phân)	Input t (Q3.13)	Địa chỉ OUT (Hệ 16)	Output $e^t$ (Q3.13)	Output $e^t$ (Thập phân)	Giá trị $e^t$ thực tế	Sai số so với thực tế (%)	
1	0	-4	x"8000"	20	x"0B31"	0.349731445	0.018315639	1809.468992	
2	1	-3.5	x"9000"	21	x"0B31"	0.349731445	0.030197383	1058.151488	
3	2	-3	x"A000"	22	x"0B31"	0.349731445	0.049787068	602.4543858	
4	3	-2.5	x"B000"	23	x"0B31"	0.349731445	0.082084999	326.060122	
5	4	-2	x"C000"	24	x"0B31"	0.349731445	0.135335283	158.4185269	
6	5	-1.5	x"D000"	25	x"0B31"	0.349731445	0.22313016	56.7387596	
7	6	-1	x"E000"	26	x"0BC5"	0.367797852	0.367879441	0.022178355	
8	7	-0.75	x"E800"	27	x"0F1A"	0.471923828	0.472366553	0.093724802	
9	8	-0.5	x"F000"	28	x"1369"	0.606567383	0.60653066	0.006054616	
10	9	-0.25	x"F800"	29	x"18EE"	0.779052734	0.778800783	0.032351188	
11	A	0	x"0000"	2A	x"2171"	1.045043945	1	4.504394531	
12	B	0.25	x"0800"	2B	x"2915"	1.283813477	1.284025417	0.016505914	
13	C	0.5	x"1000"	2C	x"34C2"	1.648681641	1.648721271	0.002403686	
14	D	0.75	x"1800"	2D	x"43C0"	2.1171875	2.117000017	0.008856088	
15	E	1	x"2000"	2E	x"56FB"	2.718139648	2.718281828	0.005230511	
16	F	1.125	x"2400"	2F	x"5BE5"	2.871704102	3.080216849	6.76941779	
17	10	1.25	x"2800"	30	x"5BE5"	2.871704102	3.490342957	17.72429997	
18	11	1.35	x"2B33"	31	x"5BE5"	2.871704102	3.857425531	25.55386802	
19	12	1.38	x"2C28"	32	x"5BE5"	2.871704102	3.974901627	27.75408373	

Từ bảng ta thấy giá trị kết quả mô phỏng gần đúng nhất với kết quả thực tế với giá trị  $t$  nằm trong khoảng từ  $-1$  đến  $1$ . Ngoài khoảng này, giá trị  $e^t$  mô phỏng bị sai lệch rất nhiều so với giá trị  $e^t$  thực tế. Giải thích về vấn đề sai số lớn với các giá trị  $t$  nằm ngoài khoảng  $[-1, 1]$  là do bản chất của thuật toán Hyperbolic CORDIC bị giới hạn bởi miền hội tụ hữu hạn ( $S_N$ ). Tổng các góc quay cơ sở  $\text{atanh}(2^{-k})$  trong thuật toán chỉ cho phép hội tụ khi giá trị đầu vào  $|t| \leq S_N$ . Với cấu hình  $N = 15$  lần lặp, miền hội tụ được xác định là  $|t| \lesssim 1.055$ . Khi  $t$  vượt quá ngưỡng này (như các giá trị  $t = -4, -3, -2$  hoặc  $t = 1.125, 1.25, 1.38$  trong bảng), thuật toán rơi vào trạng thái bão hòa (saturation). Lúc này, dù đã thực hiện hết các bước lặp, thuật toán vẫn không thể đưa biến góc mục tiêu về 0, dẫn đến kết quả đầu ra bị giữ nguyên tại các giá trị biên cố định ( $e^{-S_{15}} \approx 0.35$  và  $e^{S_{15}} \approx 2.87$ ), tạo ra khoảng cách sai số rất lớn như quan sát thấy trong cột Sai số so với thực tế (%).

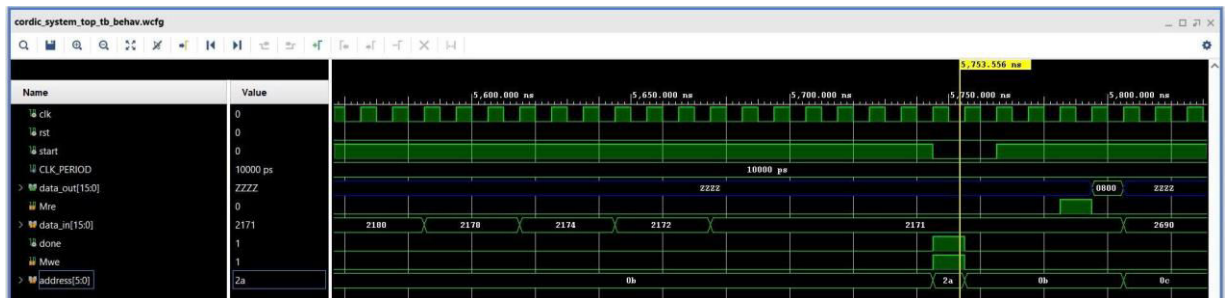
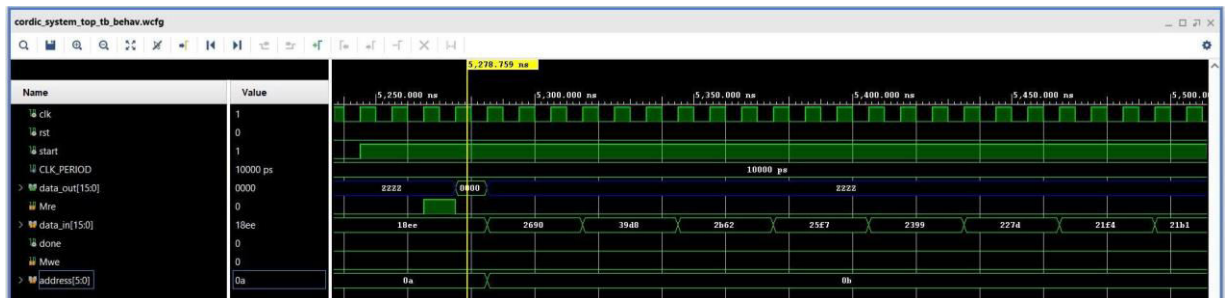
## 7. Đánh giá kết quả :

Sử dụng Module test bench ở trên với phần mềm Vivado Design Suite, ta được kết quả:

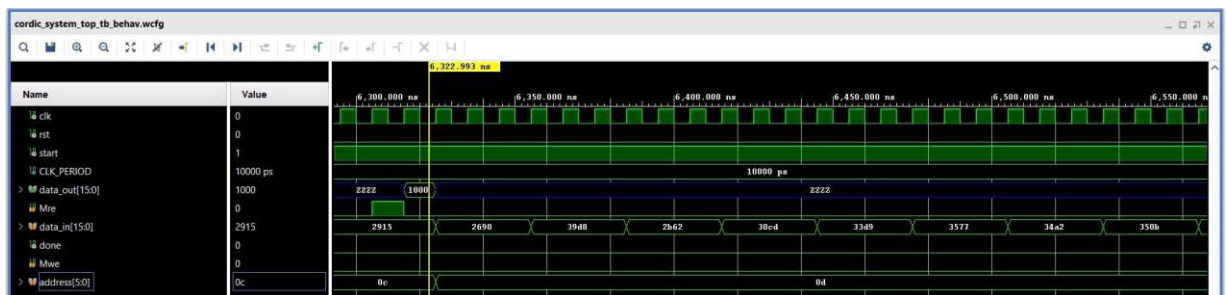
- Với đầu vào  $t = F000(-0,5)$ , ta được kết quả:

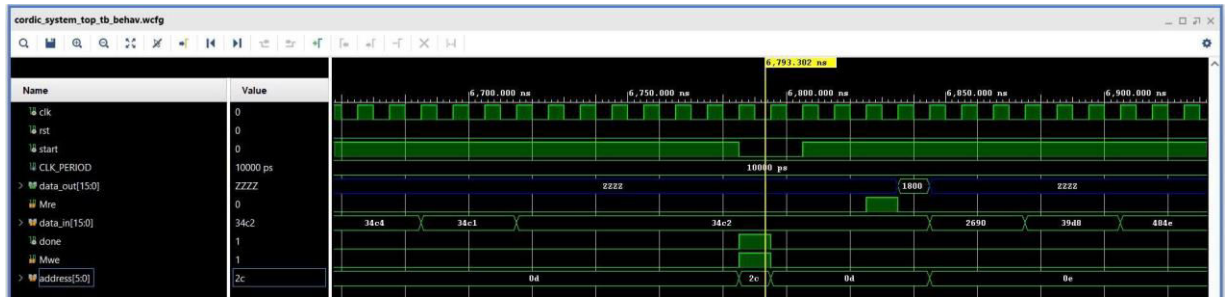


- Với đầu vào  $t = 0000$ , ta được kết quả 2171



- Với đầu vào  $t = 1000$  (0.5), ta được kết quả 34C2





Từ kết quả trên, ta thấy phần mềm Vivado Design Suite cho kết quả giống hệt ModelSim

### Ưu điểm:

- Dễ quan sát, chạy được trên các phần mềm mô phỏng khác nhau
- Code chia làm các component chuyên nghiệp, bóc tách không bị dính quá nhiều code trong 1 file
- Đã thêm tính năng memory buffer cho mạch, giúp giảm số lượng đọc ghi cho CPU

### Nhược điểm:

- Tính năng memory thêm vào vẫn chưa hoàn thiện: tín hiệu done xuất ra mỗi khi hoàn thành phép tính có thể chưa bóc tách, khiến CPU khó phân biệt ô nhớ nào đã được thực hiện xong
- Không tính toán chính xác giá trị đầu vào ngoài khoảng (-1,1)
- Sử dụng fix-point nên thiếu tính linh hoạt

## 8. Kết luận:

Dự án đã hoàn thành việc thiết kế và hiện thực hóa thành công vi mạch số tính toán hàm mũ  $e^t$  dựa trên thuật toán CORDIC Hyperbolic. Quá trình thực hiện từ nghiên cứu lý thuyết, xây dựng thuật toán đến thiết kế phần cứng bằng ngôn ngữ VHDL đã đạt được các mục tiêu đề ra ban đầu:

- Khối ExpApprox có giao diện ghép nối tới CPU sao cho CPU có thể viết giá trị đầu vào cần tính (giá trị  $t$ ) tới nó
- CPU kích hoạt quá trình tính toán của khối ExpApprox bằng cách đặt tín hiệu Start = '1'.
- Sau khi quá trình tính  $\exp(t)$  hoàn thành, khối ExpApprox sẽ báo cho CPU biết bằng cách đặt tín hiệu Done = '1';

Đồng thời đã có cải tiến về Memory\_Buffer, tuy nhiên vẫn còn 1 số hạn chế về phạm vi tính toán và tính năng thêm vào vẫn chưa hoàn thiện, sẽ cải tiến trong tương lai.

## Appendix A: Reference

[1].

---

## **Appendix B: VHDL Code**

**(đóng gói thành tệp nén và gửi kèm báo cáo)**

## Appendix C:

Compress and email to [kiemhung@vnu.edu.vn](mailto:kiemhung@vnu.edu.vn)

## List of Figures

Hình 1. Thuật toán tính xấp xỉ hàm $y = e^x$ .....	8
Hình 2. Giao diện ghép nối I/O. ....	8
Hình 3: Mô hình máy FSMD. ....	9
Hình 4: Cấu trúc của đơn vị xử lý dữ liệu Datapath. ....	9
Hình 5: Máy FSM của đơn vị điều khiển. ....	10
Hình 5: Sơ đồ khối tổng thể của bộ tính ảnh tích phân. ....	10

## List of Tables

Bảng 1: Mô tả các tín hiệu vào ra.....	7
--	---

## References

[1]