BỘ GIÁO DỤC VÀ ĐÀO TẠO
TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI

NGUYỄN MẠNH HÙNG

NGHIÊN CỨU MẠNG NEURON NHÂN TẠO VÀ THỰC HIỆN MÔ HÌNH NN TRÊN FPGA

Chuyên ngành : ĐIỆN TỬ TIN HỌC

LUẬN VĂN THẠC SĨ KHOA HỌC
ĐIỆN TỬ TIN HỌC

NGƯỜI HƯỚNG DẪN KHOA HỌC :
TS. PHẠM NGỌC NAM

Hà Nội – 2011

MỤC LỤC

LỜI CAM ĐOAN

Tôi là Nguyễn Mạnh Hùng, tôi xin cam đoan luận văn thạc sĩ điện tử tin học này do chính tôi nghiên cứu và thực hiện. Các thông tin, số liệu được sử dụng trong luận văn là trung thực và chính xác.

Hà Nội, ngày 20 tháng 09 năm 2011

Nguyễn Mạnh Hùng

LỜI CẢM ƠN

Tôi xin trân trọng cảm ơn các thầy cô giảng viên trường Đại Học Bách Khoa - Hà Nội đã truyền đạt những kiến thức quý báu cho tôi trong thời gian học cao học tại trường.

Tôi cũng xin chân thành cảm ơn TS. Phạm Ngọc Nam - Khoa Điện Tử Viễn Thông - Đại Học Bách Khoa – Hà Nội đã hướng dẫn tôi hoàn thành tốt luận văn này.

Tôi cũng cảm ơn gia đình, bạn bè đã hỗ trợ và giúp đỡ tôi hoàn thành luận văn.

Nguyễn Mạnh Hùng

DANH MỤC CÁC KÍ HIỆU, CÁC CHỮ VIẾT TẮT

Chữ viết tắt	Chữ đầy đủ
ANN/ NN	Artificial Neural Network/ Neural Network
FPGA	Field Programmable Gate Array
ASIC	Aplication Specific Integrated Circuit
CB	Cell-Based
CLB	Configurable Logic Block
DSP	Digital Signal Processing
EDA	Electronic Design Automation
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
IP	Intellectual Property
LUT	LookUp Table
VHDL	Very High Speed Integrated Circuit Hardware Description Language
CPLD	Complex Programmable Logic Devices
MAC	Mutiply ACcumlate
DCM	Digital Clock Manager
RTL	Register Tranfer Level
DDR	Double Data Rate
IOBs	Input/Output Blocks
CORDIC	COrdinate Rotation Digital Computer
FLP	FLoating Point
PWL	PieceWise Linear
MSB	Most Significant Bit
SRA	Shift Right Arithmetic
ES	Enable signal of Sigmoid Circuit
RQM	ReQuest signal of Multiplier Circuit
RDYM	ReaDY signal of Multiplier Circuit
ACC	ACCumulator

DANH MỤC CÁC BẢNG

DANH MỤC CÁC HÌNH VẼ, ĐỒ THỊ

PHẦN MỞ ĐẦU

Trong những năm gần đây, người ta thường nhắc đến "Trí tuệ nhân tạo" như là một phương thức mô phỏng trí thông minh của con người từ việc lưu trữ đến xử lý thông tin. Và nó thực sự đã trở thành nền tảng cho việc xây dựng các thế hệ máy thông minh hiện đại. Cũng với mục đích đó, nhưng dựa trên quan điểm nghiên cứu hoàn toàn khác, một môn khoa học đã ra đời, đó là "Lý thuyết mạng Neuron". Tiếp thu các thành tựu về thần kinh sinh học, mạng Neuron luôn được xây dựng thành một cấu trúc mô phỏng trực tiếp các tổ chức thần kinh trong bộ não con người.

Từ những nghiên cứu sơ khai của McCulloch và Pitts trong những năm 40 của thế kỷ trước, trải qua nhiều năm phát triển, cho đến thập kỷ này, khi trình độ phần cứng và phần mềm đã đủ mạnh cho phép cài đặt những ứng dụng phức tạp. Lý thuyết mạng Neuron mới thực sự được chú ý và nhanh chóng trở thành một hướng nghiên cứu đầy triển vọng trong mục đích xây dựng các máy thông minh tiến gần tới trí tuệ con người. Sức mạnh thuộc về bản chất tính toán song song, chấp nhận lỗi của mạng Neuron đã được chứng minh thông qua nhiều ứng dụng trong thực tiễn, đặc biệt là khi tích hợp cùng với các kỹ thuật khác.

Đa số các ứng dụng hiện nay dùng mạng Neuron nhân tạo dưới dạng các phần mềm. Tuy nhiên, có một lợi điểm trong chính cấu trúc của mạng Neuron là tính song song vốn có của chúng, do đó rất phù hợp nếu thực hiện chúng bằng phần cứng. Bên cạnh đó, FPGA là loại chip khả cấu hình, phù hợp cho các ứng dụng linh hoạt về cấu hình phần cứng, đòi hỏi xử lý song song và thời gian thực hiện ngắn. Do vậy tác giả luận văn muốn nghiên cứu và triển khai một mô hình Neuron Network (NN) trên FPGA.

Mục đích nghiên cứu của luận văn là hiện thực hóa một cấu trúc mạng Neuron trên FPGA, xây dựng mô hình mạng Neuron trên phần cứng FPGA để làm cơ sở cho việc hiện thực hóa các giải thuật huấn luyện cho mạng Neuron trên chip, và từ đó có thể mở ra các hướng thiết kế các ứng dụng xử lý thông minh trên chip.

Đối tượng nghiên cứu của luận văn bao gồm các mô hình NN lý thuyết đã có, các triển khai mô hình NN với các công cụ của các hãng nghiên cứu, các triển khai mô hình NN trên VLSI và FPGA.

Để thực hiện được mục đích nghiên cứu nêu trên, phương pháp nghiên cứu sử dụng trong luận văn là lập kế hoạch nghiên cứu chi tiết, rõ ràng trước khi bắt tay vào thực hiện nghiên cứu. Bên cạnh đó là thu thập tài liệu từ nhiều nguồn thông tin bao gồm Internet, sách báo và những người có kinh nghiệm. Đồng thời thực hiện nghiên cứu gán liền với thực nghiệm trên các công cụ thiết kế để quan sát được kết quả mô phỏng với mô hình đang nghiên cứu.

Toàn bộ nội dung luận văn được trình bày trong 4 chương với nội dung tóm tắt như sau:

Chương 1 - Tổng quan mạng Neuron nhân tạo: Nội dung này nêu tổng quan về mạng Neuron nhân tạo, các chế độ học, các luật học, cấu trúc mạng, mô hình toán học của Neuron và hàm kích hoạt.

Chương 2 - Khái quát về FPGA và việc phần cứng hóa mạng: Nội dung này nêu kiến trúc chung của FPGA và một số vấn đề liên quan. Đặc biệt với nền tảng là FPGA của Xilinx sử dụng trong luận văn. Đồng thời khái quát cách triển khai phần cứng cho mạng Neuron trên nền FPGA.

Chương 3 - Thuật toán CORDIC: Nội dung này trình bày cơ sở lý thuyết thuật toán, các kiểu tính toán, đặc điểm cũng như tính năng của thuật toán

Chương 4 - Thực hiện Neural Network trên FPGA: Nội dung này nêu lựa chọn một mô hình NN thích hợp cho việc triển khai trên phần cứng FPGA. Phần thiết kế được mô tả chi tiết với các bản thiết kế sơ đồ khối và mô tả chức năng của từng thành phần trong mô hình NN này. Đồng thời trình bày về các kết quả mô phỏng để kiểm tra chức năng hoạt động và đo đạc kết quả NN đã triển khai.

Từ kết quả nghiên cứu và thực hiện luận văn, học viên đã nghiên cứu và nắm những kiến thức về các mô hình NN đã có cũng như các triển khai thực tế của NN trên FPGA của các một số hãng nghiên cứu. Học viên đồng thời đánh giá và lựa chọn một mô hình NN phù hợp để triển khai trên FPGA, thực hiện triển khai thực tế

trên FPGA. Kết quả nghiên cứu này sẽ là cơ sở phát triển cho những nghiên cứu tiếp theo liên quan đến mạng Neuron nhân tạo và công nghệ VLSI, FPGA trong tương lai.

CHƯƠNG 1: TỔNG QUAN MẠNG NEURON NHÂN TẠO

Artificial Neural Networks (ANNs) ám chỉ các hệ thống tính toán có thành phần chủ đạo trung tâm tương tự với các mạng nơ-ron sinh học. Các mạng nơ-ron nhân tạo cũng được biết đến như "neural nets", "artificial neural systems", "parallel distributed processing systems" và "connectionist systems". Nguồn gốc của tất cả các hoạt động của mạng nơ-ron là việc nghiên cứu hệ thần kinh trước đây một thế kỉ. Nhiều thập kỉ qua, các nhà sinh học đã nghiên cứu, suy xét được một cách chính xác rằng hệ thần kinh hoạt động như thế nào. Báo cáo của William James (1890) đặc biệt xuất sắc, và mang lại nhiều vấn đề tiếp theo cho các nhà nghiên cứu sau này. Trong một mạng nơ-ron, mỗi node biểu diễn một số phép toán đơn giản, và mỗi kết nối truyền một tín hiệu từ node này tới node khác, được gắn nhãn đặt tên bởi một số hiệu gọi là "connection strength" hoặc là "weight" biểu thị phạm vi tới một tín hiệu được khuếch đại hoặc bị giảm bớt bởi một kết nối [15]

Mạng Nơ-ron nhân tạo có thể giải quyết rất nhiều vấn đề trong kĩ thuật như hệ thống lấy mẫu và điều khiển, nhận dạng đối tượng, xử lý hình ảnh, chẩn đoán y học, … Mạng nơ-ron nhân tạo cũng giống với mạng nơ-ron sinh học là các hệ thống xử lý thông tin phân tán và song song. Các hệ thống này cần thiết cho những tính toán song song quy mô lớn. Bởi vậy, thao tác tốc độ cao trong các ứng dụng thời gian thực có thể chỉ đạt được nếu các mạng được triển khai sử dụng kiến trúc phần cứng song song. Hầu hết công việc được thực hiện trong lĩnh vực này cho đến nay là các mô phỏng phần mềm, nghiên cứu tỉ mỉ những khả năng của các mô hình ANN hoặc các thuật toán mới. Nhưng việc triển khai phần cứng cũng cần thiết cho tính khả dụng và những tiến bộ trong tính song song vốn của của mạng nơ-ron. Các kiến trúc hệ thống tương tự, số và cả hỗn hợp đều được đưa ra cho việc triển khai ANNs. Kiến trúc tương tự chính xác hơn nhưng khó thực hiện và có những vấn đề với việc lưu giữ weight. Việc triển khai ANNs rơi vào hai trường hợp: Triển khai phần mềm và triển khai phần cứng. ANNs được thực hiện triển khai, huấn luyện và mô phỏng trên các máy tính tuần tự như là phần mềm để đánh giá một loạt nhiều các mô hình

mạng nơ-ron. Việc triển khai phần mềm thực hiện linh hoạt. Tuy nhiên, việc triển khai phần cứng cần thiết cho tính khả dụng và đạt được các lợi thế về tính song song vốn có của ANN.

1.1. Các đặc điểm của ANN

1.1.1. Ánh xạ đầu vào–đầu ra

Mạng được thực hiện với một bộ đầu vào và các synaptic weight của mạng được điều chỉnh để tối thiểu sự khác nhau giữa đáp ứng thực tế và đáp ứng mong muốn. Việc huấn luyện mạng được lặp đi lặp lại cho đến khi mạng đạt đến một trạng thái bền vững. Do đó mạng học từ các ví dụ bằng việc xây dựng một phép ánh xạ đầu vào-đầu ra.

1.1.2. Tính thích ứng

Các mạng nơ-ron có khả năng tích hợp bên trong để tiếp hợp các synaptic weight vào với các thay đổi môi trường xung quanh. Đặc biệt một mạng nơ-ron huấn luyện thao tác trong một môi trường cụ thể có thể được huấn luyện lại dễ dàng để xử lý các thay đổi nhỏ trong các điều kiện môi trường hoạt động. Một mạng nơ-ron có thể được thiết kế để thay đổi các synaptic weight theo thời gian thực.

1.1.3. Đáp ứng tin cậy

Một mạng nơ-ron có thể được thiết kế để cung cấp thông tin không chỉ tới đầu ra mà còn cả sự tin cậy. Thông tin này có thể được sử dụng để loại bỏ các đầu ra nhập nhằng mơ hồ do đó cải thiện sự phân loại của mạng.

1.1.4. Khả năng kháng lỗi

Một mạng nơ-ron được triển khai phần cứng có khả năng kháng lỗi vốn có. Đó là đặc tính suy giảm từ từ dưới các điều kiện hoạt động bất lợi. Do vậy, theo nguyên tắc, một mạng nơ-ron biểu lộ một sự xuống cấp từ từ theo đặc tính hơn là thất bại nặng nề.

1.1.5. Tính tương đồng trong phân tích và thiết kế

Các nơ-ron theo dạng này hay dạng khác, trình bày từ một thành phần chung chung cho đến toàn bộ các mạng nơ-ron. Tính tương đồng này làm cho nó có khả năng chia sẻ các nguyên lý và các thuật toán học trong các ứng dụng khác nhau của các mạng nơ-ron. Các mạng từng phần có thể được xây dựng bằng một sự tích hợp các khối liền với nhau.

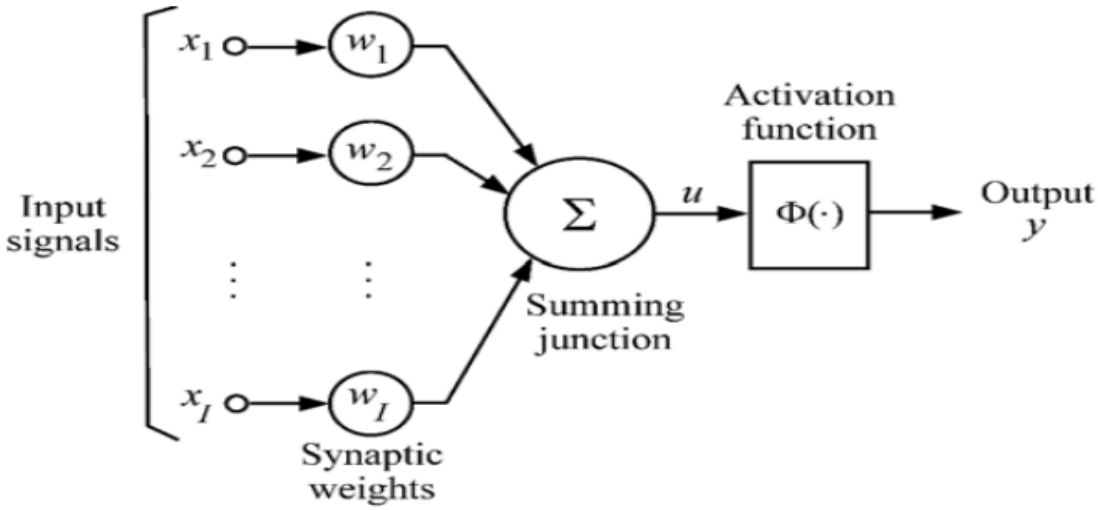
1.1.6. Tính tương đồng sinh học thần kinh

Thiết kế của một mạng nơ-ron được thúc đẩy bởi sự tương tự với bộ não, là một bằng chứng sống mà quá trình xử lý song song, khả năng kháng lỗi không chỉ theo như tự nhiên mà còn nhanh và mạnh mẽ. Các nhà kĩ thuật tìm kiếm những ý tưởng mới trong ngành sinh học thần kinh để giải quyết những vấn đề phức tạp hơn những vấn đề cơ bản trong các công nghệ thiết kế cố định thông thường.

1.1.7. Cơ chế song song

Các mạng nơ-ron có các hệ thống xử lý thông tin phân tán và song song với nhau. Do vậy các nơ-ron trong cùng một lớp có thể xử lý thông tin một cách đồng thời, cùng một lúc. Do đó các hệ thống mạng nơ-ron thực hiện nhanh hơn so với các kiến trúc tính toán khác.

1.2. Mô hình toán học của nơ-ron



Hình 1.1: Mô hình toán học của Neuron

Khi tạo một mô hình chức năng của nơ-ron sinh học, có ba thành phần cơ bản rất quan trọng. Thứ nhất, các synapse của nơ-ron là các weight. Độ bền của kết nối giữa một đầu vào và một nơ-ron được ghi chú bởi giá trị của weight. Giá trị weight âm phản ánh các kết nối để hạn chế, trong khi các giá trị dương định rõ các kết nối kích thích. Hai thành phần tiếp theo là hoạt động thực sự bên trong tế bào nơ-ron. Một bộ cộng tính tổng của tất cả các đầu vào được điều chỉnh bởi các weight tương ứng của chúng. Hoạt động này muốn nói đến sự kết hợp tuyến tính. Cuối cùng, một hàm kích hoạt điều khiển biên độ của đầu ra của nơ-ron. Một dải chấp nhận được của đầu ra luôn nằm giữa 0 và 1 hoặc -1 và 1.

Từ mô hình này, hoạt động của nơ-ron có thể được biểu thị như sau:

$$v_k = \sum_{j=1}^{p} w_{kj} x_j$$

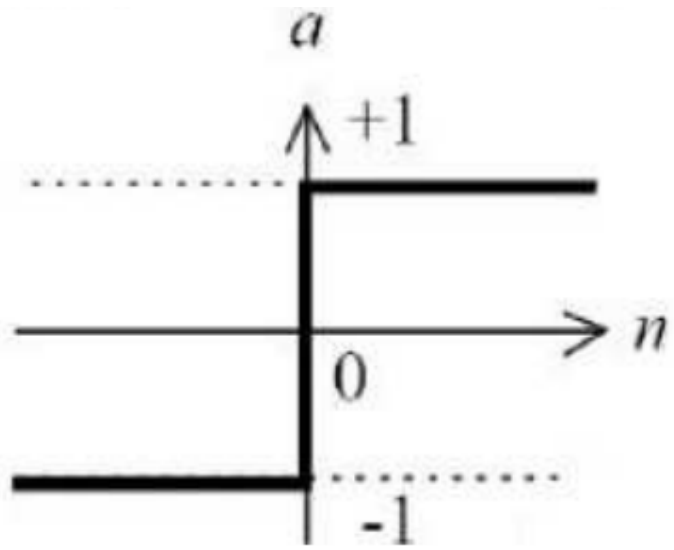
Đầu ra của nơ-ron, y_k, sẽ là kết quả qua hàm kích hoạt khi đưa giá trị của v_k vào.

1.3. Các hàm kích hoạt

1.3.1. Hàm Symmetrical Hard Limit

Hàm truyền Symmetric Hard Limit được biết với thuật ngữ "hardlims" trong matlab. Nó được sử dụng để phân loại đầu vào thành hai tập khác nhau, và có thể được định nghĩa như sau:

$$a = \begin{cases} -1 \text{ với } n < 0 \\ 1 \text{ với } n \geq 0 \end{cases}$$

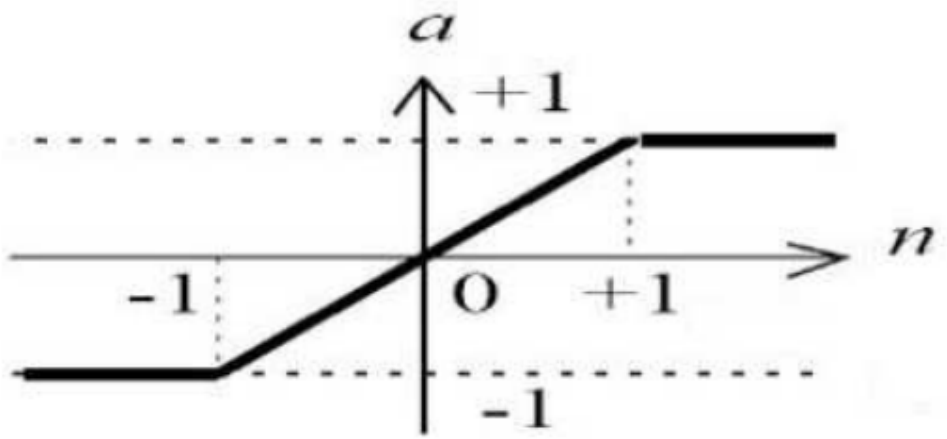


Hình 1.2: Hàm Symmetrical Hard Limit

1.3.2. Hàm Saturating Linear

Đầu ra của hàm Saturating Linear "satlins" có thể được định nghĩa như sau:

$$a = \begin{cases} -1 \text{ với } n < -1 \\ n \text{ với } -1 \leq n \leq 1 \\ 1 \text{ với } n > 1 \end{cases}$$

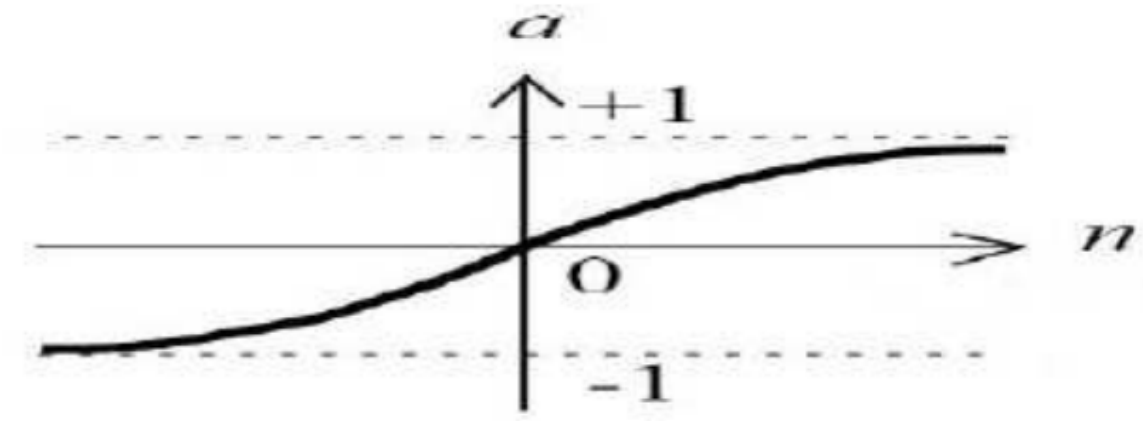


Hình 1.3: Hàm Saturating Linear

1.3.3. Hàm Hyperbolic Tangent Sigmoid

Hàm này đưa đầu vào (có giá trị bất kì trong khoảng âm vô cùng và dương vô cùng) và giá trị của đầu ra nằmg trong dải -1 đến 1, theo biểu thức sau:

$$a - \frac{e^{n} - e^{-n}}{e^{n} + e^{-n}}$$

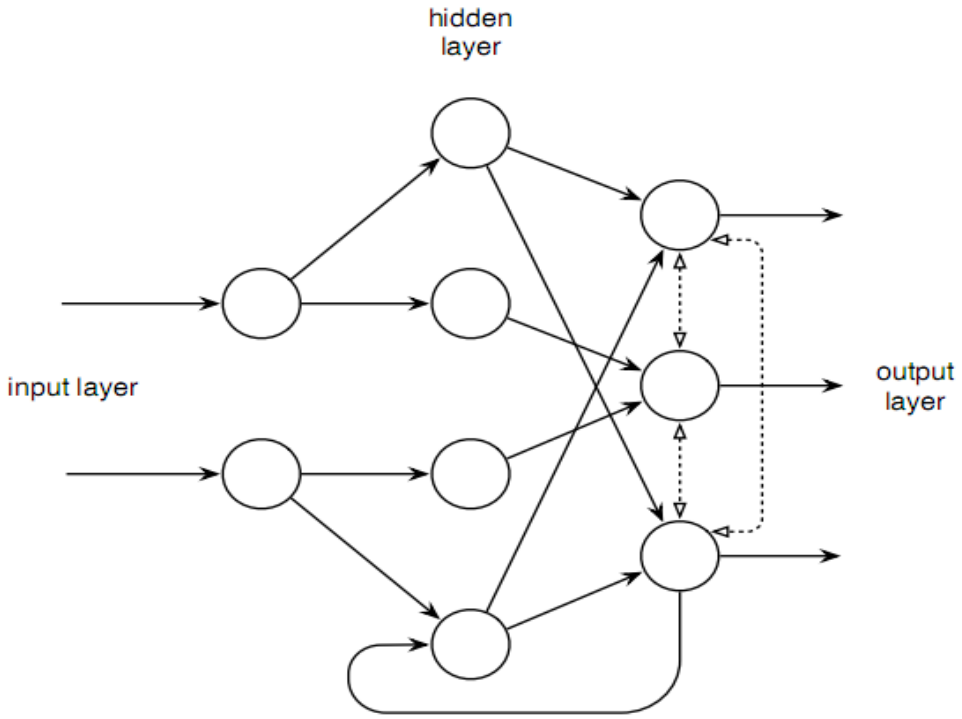


Hình 1.4: Hàm Hyperbolic Tangent Sigmoid

1.4. Cấu trúc mạng nơ-ron nhân tạo

Các mạng nơ-ron nhân tạo có chức năng như các mạng tính toán phân tán song song. Mỗi node trong mạng là một nơ-ron nhân tạo. Các nơ-ron này được kết

nối với nhau trong các cấu trúc đa dạng cho các loại vấn đề cụ thể. Điều quan trọng đáng chú ý là hầu hết các hàm cơ bản của bất kì mạng nơ-ron nhân tạo là kiến trúc của nó. Kiến trúc, cùng với thuật toán cập nhật các weight đầu vào của các nơ-ron cụ thể xác định hành vi của mạng nơ-ron nhân tạo. Điển hình là các nơ-ron tổ chức lại thành các lớp với các kết nối giữa các nơ-ron tồn tại qua các lớp, chứ không phải trong cùng một lớp với nhau. Mỗi nơ-ron trong cùng một lớp thường được kết nối đầy đủ tới tất cả các nơ-ron trong lớp liên hợp. Điều này có thể dẫn đến một số lượng cực lớn các kết nối tồn tại trong mạng, thậm chí với khá ít nơ-ron mỗi lớp. Hình 2.2 cho thấy một mạng nơ-ron đơn giản bao gồm ba lớp. Trong trường hợp này các lớp không có kết nối đầy đủ.



Hình 1.5: Một mạng nơ-ron nhân tạo cơ bản

1.4.1. Lớp đầu vào

Các nơ-ron riêng lẻ được sử dụng cho mỗi đầu vào của một mạng nơ-ron nhân tạo. Những đầu vào này có thể được thu thập dữ liệu, hoặc các đầu vào thế giới thực từ các bộ cảm biến vật lý. Quá trình tiền xử lý của các đầu vào có thể được thực

hiện để tăng tốc độ quá trình học của mạng. Nếu các đầu vào là dữ liệu thô đơn giản, thì mạng sau đó sẽ cần học để xử lý chính dữ liệu đó, cũng như phân tích nó. Điều này sẽ cần nhiều thời gian, và có thể mạng lớn hơn so với các đầu vào được xử lý.

1.4.2. Các lớp ẩn

Lớp đầu vào kết nối tới một lớp ẩn. Có thể tồn tại nhiều lớp ẩn, với nhiều đầu vào cho mỗi nơ-ron lớp ẩn thường được kết nối đầy đủ tới các đầu ra của các nơ-ron lớp trước đó. Các lớp ẩn được đưa ra đúng với ý nghĩa tên gọi do thực sự chúng không nhìn thấy các đầu vào nào và chúng đưa đến bất kì đầu ra. Chúng được đưa ra bởi các đầu ra của lớp đầu vào và dẫn tới các đầu vào của lớp đầu ra. Số lượng các nơ-ron trong mỗi lớp ẩn, cũng như số lượng chính các lớp ẩn xác định sự phức tạp của hệ thống. Việc lựa chọn chính xác số lượng cho mỗi mạng là một phần chính rất quan trọng của việc thiết kế một mạng nơ-ron làm việc. Hình 2.2 cho thấy một lớp ẩn nhưng các kiến trúc mạng khác có thể có nhiều lớp ẩn.

1.4.3. Lớp đầu ra

Mỗi nơ-ron trong lớp đầu ra nhận đầu ra của mỗi nơ-ron trong lớp ẩn cuối cùng. Lớp đầu ra cung cấp các đầu ra thực sự của hệ thống đó. Những đầu ra này có thể đưa tới quá trình xử lý tính toán khác, một hệ thống điều khiển cơ khí, hoặc có thể được lưu trong một file để phân tích. Giống như hàm đầu ra của một nơ-ron, lớp đầu ra có thể tham gia vào một số loại cạnh tranh giữa các đầu vào. Sự hạn chế ở bên có thể được nhìn thấy ở hình 2.2 như là đường nhiều chấm kết nối các nơ-ron đầu ra. Thêm vào đó, các đầu ra có thể cũng được phản hồi lại các nơ-ron trước đó để hỗ trợ quá trình học. Trong hình 2.2, kết quả từ một nơ-ron đầu ra được phản hồi vào một nơ-ron trong lớp ẩn.

1.5. Các chế độ học

Nhiều chế độ học khác nhau cho việc xác định các weight của các nơ-ron riêng lẻ được cập nhật trong một mạng như thế nào và khi nào. Các loại học hoặc là

học có giám sát hoặc là học không có giám sát. Như những phát biểu trước đây, học không có giám sát là loại học không được biết nhất hiện nay.

1.5.1. Học có giám sát

Quá trình học trong một chế độ giám sát bắt đầu với sự so sánh các đầu ra được sinh ra của mạng với các đầu ra mong muốn. Các weight đầu vào của mỗi nơ-ron được điều chỉnh để tối thiểu hóa các sai khác được tìm thấy. Quá trình này được lặp đi lặp lại cho đến khi mạng được cho rằng là tích lũy đủ. Sau pha huấn luyện, các weight của các nơ-ron cố định lại, cho phép mạng được sử dụng một cách đáng tin cậy. Thích ứng tốt hơn để ít biến đổi, tốc độ học có thể thấp hơn. Một trong các thứ quan trọng khác để làm khi huấn luyện một mạng là lựa chọn cẩn thận dữ liệu để huấn luyện. Điển hình là dữ liệu được tách thành một bộ huấn luyện và một bộ thử nhỏ hơn rất nhiều. Bộ huấn luyện được sử dụng để huấn luyện mạng để thực hiện nhiệm vụ. Bộ thử được sử dụng để xác thực lại mạng có thể khái quát hóa cái nó đã học để ít biến đổi. Không có việc phân tách bộ dữ liệu, người ta sẽ không thể biết mạng ghi nhớ bộ dữ liệu đơn giản hay không.

1.5.2. Học không giám sát

Học không giám sát được thực hiện không có bất cứ hình thức tăng cường nào bên ngoài. Chế độ học này thể hiện một loại mục đích cuối cùng cho những người thiết kế hệ thống. Sử dụng phương pháp này, tự hệ thống dạy chính nó. Mạng bao gồm trong nó một phương pháp xác định khi các đầu ra của nó không phải cái chúng sẽ đạt được. Phương pháp học này không được biết nhiều như phương pháp học có giám sát. Nó yêu cầu mạng học trực tuyến. Cơ cấu hiện tại bị giới hạn bởi sơ đồ tự tổ chức, học để ohân lọa dữ liệu đi vào. Các phát triển xa hơn sau này với loại học này sẽ sử dụng trong nhiều tình huống mà sự thích ứng với các đầu vào mới cần đều đặn.

1.6. Tốc độ học

Tốc độ học của một mạng được xác định bởi nhiều yếu tố. Kiến trúc mạng, kích cỡ và độ phức tạp có một vai trò lớn trong tốc độ mà mạng học. Một nhân tố khác ảnh hưởng đến tốc độ học là luật học hoặc các luật triển khai. Tốc độ học nhanh hay chậm có những lí do riêng của chúng. Và tốc độ học của một mạng ảnh hưởng mạnh mẽ tới hiệu năng của nó.

Các luật học chi phối cách weight đầu vào của nơ-ron trong mạng được điều chỉnh. Cụ thể lỗi ở đầu ra được truyền ngược qua nhiều lớp mạng, tùy chỉnh các weight khi nó đến. Lỗi được truyền ngược như thế nào là sự khác nhau chính. Theo [10], là một số luật hay được sử dụng bởi các kiến trúc mạng.

1.7. Các luật học phổ biến

1.7.1. Luật Hebb's

Luật Hebb's là luật phổ biến đầu tiên để cập nhật weight. Nó thực hiện như sau "Nếu một nơ-ron nhận một đầu vào từ một nơ-ron khác, và nếu cả 2 đều tích cực cao (về mặt toán học là cùng dấu), weight giữa các nơ-ron sẽ được tăng cường". Hebb đã quan sát rằng các quá trình nơ-ron sinh học được củng cố mỗi lần chúng được sử dụng, và luật này được thiết kế để mô phỏng hiệu quả. Hầy hết các luật khác xây dựng dựa trên nguyên lý luật này.

Như là một ví dụ cho cách luật này hoạt động, giả định một mạng nơ-ron được huấn luyện để điều khiển gia tốc của một xe ô tô. Giả định hơn nữa là các đầu vào của mạng là thắng phanh xe và vị trí bàn đạp ga được điều khiển hoạt động bởi một người lái xe. Sự gia tốc và giảm tốc của xe có thể được so sánh với đầu ra mong muốn của người lái xe. Bây giờ giả sử người lái xe của xe ô tô đó muốn đi chậm lại và kéo thắng phanh. Nếu đầu ra của mạng nơ-ron được giảm tốc, bất kì weight đầu vào, một câu lệnh "giảm tốc" sẽ được đưa ra, sẽ tăng lên. Điều này sẽ tăng cường tích cực hành vi chấp nhận của mạng.

1.7.2. Luật Delta

Luật Delta là một trong các luật học phổ biến nhất, và là một sự biến đổi của luật Hebb's. Nó cũng được biết bởi một số tên khác, gồm có tên luật học Widrow-Hoff và luật học Trung Bình Bình Phương Tối Thiểu (Least Mean Square Learning Rule). Nó làm việc bằng cách biến đổi lỗi ở đầu ra bởi đạo hàm của hàm truyền đạt. Kết quả của phép biến đổi này được sử dụng để điều chỉnh các weight đầu vào kết hợp với các đầu ra của các lớp trước đó. Kết quả lỗi biến đổi được truyền ngược qua tất cả các lớp. Các mạng truyền thẳng, truyền ngược sử dụng phương pháp học này [15]

1.7.3. Luật giảm Gradient

Luật Gradient Descent giống như luật Delta trong phép lấy đạo hàm hàm truyền đạt điều chỉnh lỗi đầu ra. Một hằng số tỉ lệ cộng thêm có quan hệ với tốc độ học được thêm vào yếu tố điều chỉnh trước khi các weight được điều chỉnh. Phương pháp này cơ bản được biết là có tốc độ hội tụ thấp.

1.7.4. Luật học Kohonen

Luật học này được sử dụng cho mạng không giám sát. Teuvo Kohonen lấy cảm hứng từ quá trình học trong các hệ thống sinh học, và do đó đưa ra luật này. Với luật Kohonen, các nơ-ron cạnh tranh cơ hội học. Mạng với đầu ra lớn nhất sẽ chiến thắng, và đi tới cập nhật weight của nó và có thể một số lân cận của nó.

1.8. Quá trình dạy Back-Propagation

Có một số phương pháp khác nhau để thiết lập các synaptic weight và các giá trị ngưỡng. Phương pháp phổ biến nhất là thuật toán truyền ngược lỗi (back propagation of error). Đây là một thuật tóan học có giám sát, nghĩa là phải dạy cho mạng phản hồi như thế nào dựa trên bộ các đối tượng đầu vào cụ thể. Quá trình dạy này thực hiện theo các bước như sau:

1.Đưa ra một đối tượng đầu vào.

2.Đọc ra đối tượng đầu ra kết quả

3.So sánh đầu ra thu được với đầu ra mong muốn và sinh ra tín hiệu lỗi nếu có sự khác nhau.

4.Tín hiệu lỗi này được đưa tới các nơ-ron đầu ra và truyền qua mạng theo hướng ngược lại với hướng của các tín hiệu truyền thẳng.

5.Các weight và các ngưỡng sau đó được thay đổi dựa trên các tín hiệu lỗi này để làm giảm sự khác nhau giữa đầu ra và đích mong muốn.

Các bước này hoặc là được lặp lại theo các bước rời rạc riêng biệt hoặc là được thực hiện một cách đồng thời theo kiểu song song và liên tục cho tất cả các đối tượng đầu vào, cho đến khi mạng đáp ứng đúng chính xác. Thuật toán học có thể được biểu diễn với hai phương trình. Một phương trình đo lỗi trên đầu ra, và một phương trình biểu diễn sự thay đổi của một weight. Lỗi này luôn được đo khi sự khác nhau giữa đầu ra mong muốn hay còn gọi là đích mong muốn và đầu ra thực tế. Hàm thay đổi weight sau đó được tính tỉ lệ với đạo hàm bình phương của lỗi mỗi đối tượng đầu ra đo được đối với mỗi weight và với hằng số tỉ lệ âm. Điều này sẽ thực hiện một sự tìm kiếm giảm gradient trong bình phương lỗi cho lỗi tối thiểu [16]

Phép tính thực hiện bởi một nơ-ron trong mạng nơ-ron truyền thẳng hầu như giống với giả thuyết. Đầu ra là một hàm rõ ràng của đầu vào và được mô tả bởi:

$$o_{pj} = f_j(\sum_i w_{ji} o_{pi} + \theta_j) = f_j(net_{pj})$$

Trong đó, w_{ji} là weight của đầu vào i và nơ-ron j, o_{pi} là đầu vào i, đó là đầu ra của lớp trước, cho đối tượng đầu vào p, θ_j là giá trị ngưỡng và f_j là hàm kích hoạt cho nơ-ron j.

Cụ thể hơn:

$$E_p = \frac{1}{2}\sum_j (t_{pj} - o_{pj})^2$$

Là bình phương của lỗi đo được cho đối tượng đầu vào p, trong đó t_{pj} biểu diễn đích mong muốn của nơ-ron đầu ra j, và o_{pj} là đầu ra thực sự. Phương trình thay đổi weight được tính sau đó là:

$$\Delta_p w_{ji} = -\eta \frac{\partial E_p}{\partial w_{ji}}$$

Trong đó $\Delta_p w_{ji}$ là sự thay đổi weight và η là nhân tố tỉ lệ định nghĩa tốc độ học của thuật toán. Giải quyết sự khác nhau này có thể được thể hiện rõ trong hai phương trình, tùy thuộc vào weight cần quan tâm. Nếu weight thuộc về một nơ-ron đầu ra phép lấy vi phân là truyền thẳng và có:

$$\Delta_p w_{ji} = -\eta \frac{\partial E_p}{\partial w_{ji}} = \eta (t_{pj} - o_{pj}) f_j'(net_{pj}) o_{pi} = \eta \delta_{pj} o_{pj}$$

Trong đó f_j' là đạo hàm của hàm kích hoạt cho nơ-ron đầu ra j và o_{pi} là đầu vào i của nơ-ron này cho đối tượng p. δ chỉ là cách biểu diễn chuẩn lỗi tỉ lệ với đạo hàm của đầu ra. Nếu weight thuộc nơ-ron ẩn, người ta áp dụng qui tắc dây chuyền:

$$\Delta_p w_{ji} = -\eta \frac{\partial E_p}{\partial w_{ji}} = \eta \left(\sum_k \delta_{pk} w_{kj} \right) f_j'(net_{pj}) o_{pi} = \eta \delta_{pj} o_{pj}$$

Trong đó δ_{pk} là δ cho nơ-ron k trong lớp tiếp theo.

1.9. Triển khai FPGA

Hiện nay, hầu hết các nghiên cứu và kiểm thử mạng nơ-ron được thực hiện sử dụng các mô phỏng phần mềm. Triển khai phần mềm cho phép phân tích các hành vi mạng dễ dàng. Thêm vào đó, các vấn đề kiểu dự báo trước trong nhiều trường hợp không cần ứng dụng phần cứng nhúng. Tuy nhiên, mạng nơ-ron phần cứng có thể đạt được rất nhiều các lợi ích [12]

Một triển khai phần cứng tối ưu có thể mang lại hiệu năng tốt hơn cấu hình phần mềm chạy trên một bộ vi xử lý chuẩn. Thêm vào đó, một triển khai phần cứng có thể thực hiện cho các ứng dụng khó đạt được trong thiết lập phần mềm, như với các ứng dụng cảm nhận từ xa. Việc thiết kế cho một FPGA cũng đạt được nhiều lợi ích, thảo luận trong chương 2.

Khả năng của FPGA là khả trình có một số tính năng nổi trội cụ thể. Nhiều kĩ thuật trong việc gia tăng mật độ để tăng số lượng "chức năng mạch hiệu quả trên

mỗi đơn vị mạch". Một cách thực hiện điều này là phân tách ra các giai đoạn khác nhau của thuật toán học mạng trong nhiều vùng khác nhau của FPGA. Một cách khác là sử dụng các bộ nhân hệ số không thay đổi tối ưu để xử lý các phép tính weight của các đầu vào nơ-ron. Các phép tính này sẽ được thực hiện nhanh [3].

FPGA cũng dùng cho việc triển khai thực hiện phần cứng của các mạng nơ-ron nhân tạo, có thể điều chỉnh động đồ hình của chúng. Điều này cấp cho các thuật toán học phức tạp để điều chỉnh đồ hình, đưa đến một mạng cuối cùng tinh vi hơn.

Các triển khai phần cứng cải thiện những vấn đề không thực hiện được của các thiết kế phần mềm. Một điều đáng chú ý đó là cách trình bày số, về cơ bản có hai cách mà một số có thể được miêu tả trong bất cứ thiết bị phần cứng nào: dấu phẩy tĩnh và dấu phẩy động. CORDIC sử dụng ký hiệu dấu phẩy tĩnh là một số nguyên tỉ lệ đơn giản. Dấu phẩy động là cách miêu tả phổ biến nhất được sử dụng trong tính toán phần cứng vì dải giá trị rộng có thể thực hiện. Trong một thiết kế phần cứng, đặc biệt là một thiết kế ở đó các đơn vị số học hỗ trợ một kiến trúc phần cứng rất phức tạp (Ví dụ: mạng Neuron nhân tạo), phạm vi khu vực tính năng chip yêu cầu một đơn vị dấu phẩy động quá cao, điều đó không khả thi để triển khai một mạng Neuron trên FPGA sử dụng các weight dấu phẩy động. Các khối dấu phẩy tĩnh có nhiều hấp dẫn hơn trong việc chúng cấp thêm phạm vi khu vực tính năng chip để dành cho mạng neural thực tế. Có một lượng lớn các nghiên cứu đang thực hiện về việc sử dụng các weight dấu phẩy tĩnh trong các mạng Neuron, loại bỏ sự cần thiết của các phần cứng dấu phẩy động.

Theo việc lựa chọn cách trình bày số dẫn đến quyết định độ chính xác của các weight nên có như thế nào. Vì với bất cứ hệ thống nào, độ chính xác càng lớn đưa đến việc làm tăng thời gian tính toán, các yêu cầu phạm vi khu vực tính năng chip và nguồn tiêu thụ càng lớn. Vì bất cứ vấn đề gì, việc hi vọng loại bỏ một số vấn đề kết hợp với một độ chính xác cao, gọi là "độ chính xác tối thiểu" phải được xác định. Với một số ứng dụng nó có thể là 16 bits.

1.10. CORDIC

Bất cứ triển khai Neural Network nào cũng cần một đơn vị số học để thực hiện các tính toán cần thiết trong mạng. Các mạng Neuron lớn và dựa vào sức mạnh tính toán song song của các Neuron của chúng về mặt hiệu suất. Số lượng các Neuron lớn nghĩa là các mạng Neuron có các yêu cầu tài nguyên phần cứng đáng kể. Do vậy, chỉ các đơn vị số học rất nhỏ mới có thể được sử dụng trong việc triển khai phần cứng.

Thuật toán thỏa mãn yêu cầu này. CORDIC có khả năng tính toán nhiều hàm được sử dụng trong các mạng Neuron. Thực tế nó có thể chuyển đổi giữa các bộ hàm một cách dễ dàng nghĩa là nó có thể được sử dụng trong mạng dùng các hàm truyền đạt khác nhau trong các lớp mạng khác nhau.

Các hàm truyền đạt phổ biến nhất được sử dụng trong các mạng Neuron là hàm Sigmoid như là Hyperbolic Tangent. Khi trong mode tính toán thích hợp, CORDIC có thể được sử dụng để tính hàm này với sự giúp đỡ của một bộ chia nhị phân, sử dụng đồng nhất thức $\tanh(x)=\frac{\sinh(x)}{\cosh(x)}$. CORDIC cũng có khả năng thực hiện việc chia này khi ở trong mode vector tuyến tính, nhưng việc chia chậm và sẽ tốn nhiều thời gian tính toán hơn. CORDIC cũng có thể thực hiện phép tính nhân khi ở mode rotation tuyến tính, có thể tính các weight cho đầu vào Neuron, mặc dù các đơn vị tính nhân tối ưu cho việc nhân với một hằng số thích hợp hơn khi các weight được xác định sẵn. Luận văn này nghiên cứu việc thực hiện hàm số mũ trong Neuron Network trên nền FPGA sử dụng CORDIC. Mặc dù không phổ biến như Hyperbolic Tangent, các ứng dụng mà hàm số mũ được sử dụng như là hàm truyền đạt.

CHƯƠNG 2: KHÁI QUÁT VỀ FPGA VÀ VIỆC PHẦN CỨNG HÓA MẠNG NEURON

FPGA (Field Programmable Gate Array) có ứng dụng rất lớn trong nhiều lĩnh vực như xử lý tín hiệu số DSP, các hệ thống hàng không, vũ trụ, quốc phòng, xử lý ảnh… bởi tính linh động cao trong quá trình thiết kế, giúp người lập trình có thể xử lý những bài toán phức tạp mà trước kia chỉ thực hiện nhờ phần mềm máy tính. Chương này sẽ đề cập một cách tổng quan về lịch sử ra đời và phát triển của FPGA, cấu trúc và trình tự thiết kế trên FPGA. Tiếp theo, chương này trình bày sơ lược về hai ngôn ngữ mô tả phần cứng được dùng phổ biến hiện nay là Verilog và VHDL.

2.1. Giới thiệu chung về FPGA và ngôn ngữ HDL

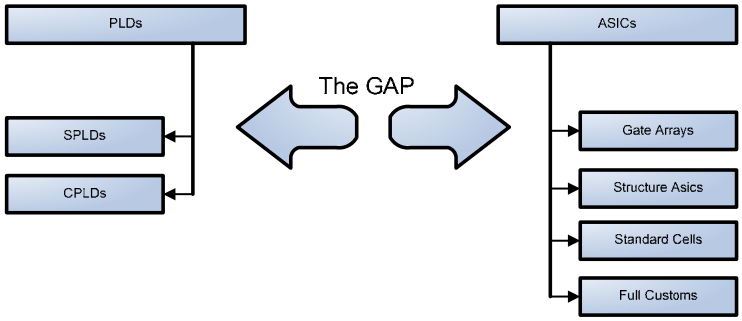
2.1.1. Khái niệm và ứng dụng FPGA

Khái niệm

Trong khoảng giữa của thập kỷ 80, sự phân tách về mặt chức năng và ứng dụng giữa các IC số ngày càng trở nên rõ nét với một bên là các thiết bị có thể lập trình được như SPLDs và CPLDs và một bên là các thiết bị chỉ dùng cho các ứng dụng chuyên biệt ASICs. Các thiết bị như SPLDs và CPLDs có khả năng mềm dẻo trong thiết kế, thời gian phát triển sản phẩm ngắn và có thể nâng cấp dễ dàng do có thể lập trình lại nhiều lần nhưng không có khả năng xử lý các bài toán có độ phức tạp cao. Ngược lại, ASICs có khả năng xử lý những phép toán đặc biệt phức tạp nhưng chi phí thiết kế lại tốn kém và tiêu tốn nhiều thời gian để đưa được một sản phẩm ra thị trường. Một ASICs như tên gọi của nó chỉ xử lý duy nhất một công việc cụ thể và không có khả năng nâng cấp như PLDs [5]

Để lấp đầy sự phân tách đó, Xilinx đã đi tiên phong trong việc nghiên cứu và phát triển một dòng IC mới được gọi là FPGA và chính thức đưa ra thị trường vào năm 1984. Công nghệ FPGA có thể được ứng dụng rộng rãi từ thiết kế các mạch logic cho đến những hệ thống lớn tích hợp vi xử lý hoặc xử lý đồ họa. Hầu hết các chip FPGA đều có thể được cấu hình lại nhiều lần do đó rất mềm dẻo, linh hoạt

trong thiết kế. Ngày nay FPGA còn được sử dụng để thiết kế và kiểm tra mẫu trước khi đưa đi chế tạo chip ASIC. Mật độ tích hợp cao của các cổng logic và các flipflop là điểm khác nhau cơ bản của FPGA so với các thiết bị lập trình được khác như SPLD hay CPLD. FPGA và CPLD đều bao gồm một số lượng khá lớn các phần tử logic khả trình. Mật độ cổng logic (Logic Gate) của CPLD nằm trong khoảng từ vài nghìn cho đến 10 nghìn cổng. Trong khi đó FPGA thông thường chứa từ 10 nghìn cho đến vài triệu cổng.



Hình 2.1: Sự phân tách FPGA và ASIC

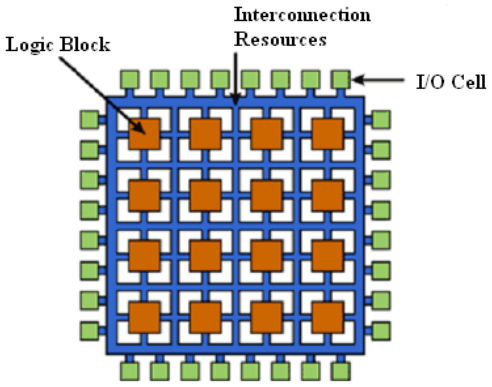
Khác biệt cơ bản giữa FPGA và CPLD là ở kiến trúc của chúng. CPLD có kiến trúc bị giới hạn trong một hoặc một vài dãy logic khả trình cùng với một lượng nhỏ thanh ghi định thời. Do đó nó kém linh hoạt hơn, nhưng lại có ưu điểm là khả năng dự đoán trễ lớn hơn và tỉ lệ logic kết nối cao hơn. Ngược lại, trong kiến trúc của FPGA lại có sự trội hơn về số lượng kết nối. Điều này làm cho nó trở nên linh hoạt hơn (về số lượng thiết kế được thực thi bên trong) nhưng cũng đồng nghĩa với việc phức tạp hơn trong quá trình thiết kế.

Một khác biệt đáng chú ý nữa giữa FPGA và CPLD là: hầu hết các FPGA hiện nay đều có các phần tử chức năng tích hợp cao hơn (như bộ cộng, nhân tích hợp và bộ nhớ tích hợp).

Một số kiến trúc FPGA hiện nay còn có thể cho phép tái cấu hình lại từng phần. Có nghĩa là cho phép một phần của thiết kế được cấu hình lại trong khi những thiết kế khác vẫn tiếp tục hoạt động. Một ưu điểm khác của FPGA là người thiết kế có thể tích hợp vào đó các bộ xử lý mềm hay vi xử lý tích hợp nhúng. Các vi xử lý này có thể được thiết kế như các khối logic thông thường, mà mã nguồn do các

hãng cung cấp thực thi các lệnh theo chương trình được nạp riêng biệt, và có các ngoại vi được thiết kế linh động (khối giao tiếp UART, vào/ra đa chức năng GPIO, Ethernet, …). Các vi xử lý này cũng có thể được tính toán khả trình lại ngay trong khi đang chạy.

FPGA đang phát triển với tốc độ chóng mặt. Các tính năng được hỗ trợ trong các dòng sản phẩm khác nhau thay đổi gần như hằng ngày. Tuy nhiên kiến trúc cơ bản của một chip FPGA vẫn bao gồm một ma trận các khối logic CLB có thể cấu hình được và các kết nối giữa chúng, phía ngoài được bao phủ bởi các khối vào ra I/O. Các CLB thường có cấu trúc phức tạp được tạo nên bởi các LUT và MUX. Hệ thống kết nối bên trong chip gồm có các đường dây ngang và dọc có thể được định nghĩa thông các chuyển mạch lập trình được.



Hình 2.2: Cấu trúc cơ bản chip FPGA

Ngoài các khối cơ bản nêu trên chip FPGA thường có thêm các thành phần khác

- Bộ nhân, bộ cộng.
- Nhân vi xử lý.
- Bộ nhớ Block Ram.
- Bộ quản lý xung đồng hồ.

Ứng dụng

FPGA được ứng dụng điển hình trong các lĩnh vực như: xử lý tín hiệu số, xử lý ảnh, thị giác máy, nhận dạng giọng nói, mã hóa, mô phỏng, ... FPGA đặc biệt

mạnh trong các lĩnh vực hoặc ứng dụng mà kiến trúc của nó yêu cầu xử lý song song, đặc biệt là mã hóa và giải mã. FPGA cũng được sử dụng trong những ứng dụng cần thực thi các thuật toán như FFT, tích chập thay thế cho vi xử lý. Hiện nay công nghệ FPGA đang được sản xuất và hỗ trợ phần mềm bởi các hãng như: Xilinx, Altera, Actel, Atmel, ... trong đó Xilinx và Altera là hai hãng hàng đầu. Xilinx cung cấp phần mềm miễn phí trên nền Windows, Linux, trong khi Altera cung cấp những công cụ miễn phí trên nền Windows, Linux và Solaris [14]

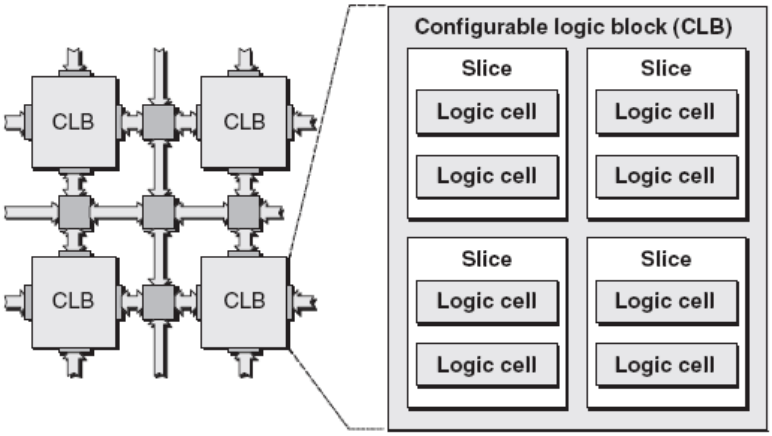
2.1.2. Kiến trúc FPGA

Khối Vào/Ra

Ngày nay một chíp FPGA có thể có hơn 1000 chân và được sắp xếp thành các từng dãy xung quanh chip. Các chân vào ra này có thể cấu hình để hoạt động ở các chuẩn khác nhau: LVTTL, LVCMOS, LVDS, … Mỗi chuẩn sẽ quy định mức điện áp khác nhau biểu diễn mức 0 hoặc 1. Việc có thể cấu hình các chuẩn I/O giúp việc thiết kế hệ thống trở nên đơn giản hơn rất nhiều. Mỗi bank I/O có thể được cấu hình với một chuẩn vào ra riêng. Do đó chip FPGA có thể sử dụng trong các hệ thống yêu cầu nhiều chuẩn tín hiệu vào ra khác nhau. Ngoài việc có thể cấu hình theo các chuẩn vào ra khác nhau các chân I/O còn có thể điều chỉnh giá trị trở kháng đặt trên nó. Ngày nay các hệ thống số thường hoạt động ở tần số rất cao (thời gian tín hiệu chuyển giữa các mức logic thường rất nhỏ), để ngăn tín hiệu phản hồi trở lại, cần phải có các điện trở thích hợp nối với các chân vào ra của FPGA. Trong quá khứ, những điện trở này thường được thiết kế trên mạch tuy nhiên khi số chân FPGA ngày càng tăng thì việc này trở nên bất tiện. Vì lý do đó ngày nay các chip FPGA sử dụng các điện trở nối bên trong chip có thể thay đổi được giá trị để phù hợp với từng điều kiện làm việc và từng chuẩn vào ra.

Các khối logic khả cấu hình CLBs

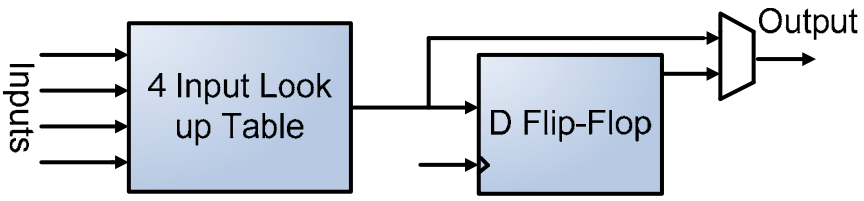
Cấu trúc cơ bản của chip FPGA bao gồm một ma trận các khối logic có thể lập trình được CLB đặt trên một hệ thống các đường kết nối và ma trận chuyển mạch. Ta có thể quan sát vị trí của CLB trong chip FPGA trong hình dưới đây:



Hình 2.3: Sơ đồ khối CLB

Ngoài các đường kết nối giữa các CLB, bên trong mỗi CLB cũng có các kết nối có thể lập trình được. Chính khả năng này giúp CLB có thể lập trình để thực hiện các hàm logic khác nhau. Trong các dòng chip cũ mỗi CLB có thể được cấu thành bởi hai slice nhưng trong các kiến trúc mới thì số slice trong mỗi CLB thường là bốn. Điều này làm tăng khả năng tính toán của mỗi CLB và của cả chip FPGA [13]

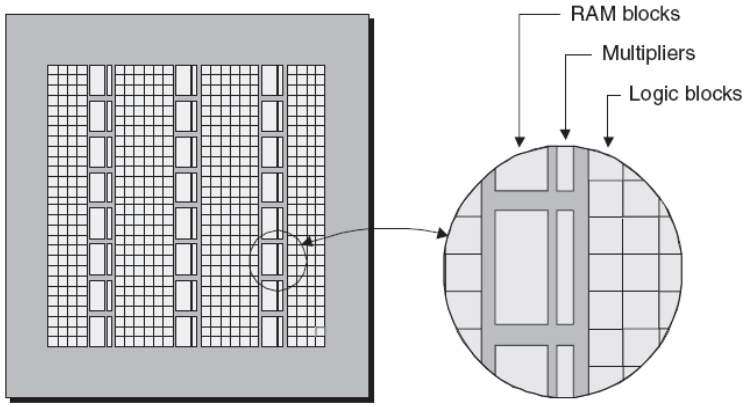
Mỗi LUT có thể thực hiện bất cứ một hàm logic bốn đầu vào nào do đó nó còn được gọi là bộ tạo hàm. LUT là một phần tử cơ bản trong thiết kế số có thể chuyển sang ASIC. Độ trễ tính toán ở trong LUT là hằng số bất kể hàm logic được thực hiện có phức tạp hay không. Một hàm logic lớn hơn bốn đầu vào được thực hiện bởi sự kết hợp của hai hoặc nhiều LUT. Tuy nhiên điều này sẽ làm độ trễ của hàm logic tăng gấp hai lần, làm giảm tốc độ của toàn bộ hệ thống. Hình 2.4 là một ví dụ thực hiện một hàm logic bằng LUT.



Hình 2.4: Một logic block điển hình

Khối RAM

Ngày nay rất nhiều ứng dụng yêu cầu phải sử dụng bộ nhớ, vì lý do đó, chíp FPGA đã được nhúng nhiều khối RAM có kích thước tương đối lớn gọi là e-RAM hay Block RAM. Tùy theo từng kiến trúc, các khối Ram này có thể được sắp xếp ở vùng biên của chip, sắp xếp rải rác trên bề mặt chip trong những vùng khác nhau hoặc được xếp thành cột như hình 2.5:



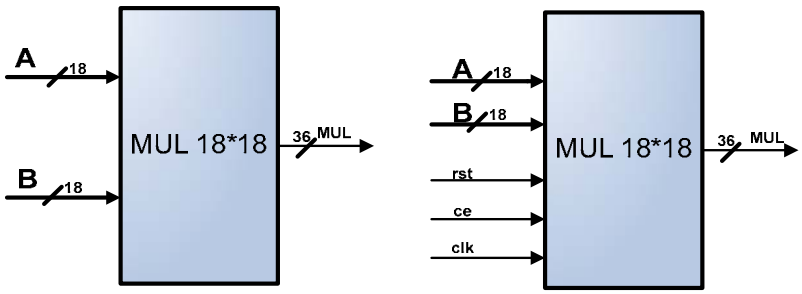
Hình 2.5: Vị trí các Block Ram

Phụ thuộc vào từng dòng chip mỗi block RAM có thể có kích thước từ vài nghìn cho đến vài chục nghìn bit và mỗi chip có thể chứa từ hàng chục cho đến hàng trăm block Ram, do đó cung cấp bộ nhớ có dung lượng từ hàng trăm nghìn bit cho đến vài triệu bit. Mỗi Block Ram có thể được sử dụng độc lập hoặc kết hợp nhiều block Ram với nhau để có kích thước lớn hơn. Những block Ram này có thể được dùng vào rất nhiều mục đích ví dụ như thực hiện các bộ nhớ một cổng, hai cổng, FIFO, FSM, …

Đối với các chip FPGA của Xilinx, BRAM thường có kích thước 16 kbit và được thiết kế với hai cổng. Điều đó có nghĩa là mỗi BRAM có một cổng đọc và một cổng ghi độc lập có xung đồng hồ riêng. Mỗi BRAM có thể thay đổi kích thước (độ rộng) các cổng để phù hợp với các ứng dụng khác nhau. Trong một vài chip của Xilinx như chip Virtex-4 các block Ram có thể hoạt động ở tần số 500MHz và có thêm các khối logic để thực hiện việc ghép các BRAM lại với nhau hoặc để sử dụng BRAM như một FIFO một cách hiệu quả nhất mà không tốn thêm một slice nào.

Các bộ nhân, cộng, MAC

Một vài phép toán, như phép nhân, nếu được thực hiện đơn thuần bằng các CLB sẽ rất tốn tài nguyên và độ trễ khi thực hiện phép toán cũng rất lớn. Các phép toán này lại được sử dụng trong rất nhiều ứng dụng, do đó trong các chip FPGA đã được nhúng thêm những khối logic riêng biệt chuyên thực hiện các phép toán này. Các khối logic này thông thường được đặt gần các block ram để giảm thời gian truyền dữ liệu vì chúng thường hay được kết hợp với nhau để giải quyết các bài toán. Trong các chip FPGA của Xilinx thường được tích hợp các bộ nhân hai đầu vào 18 bits. Các bộ nhân có thể thực hiện các phép nhân có dấu hoặc không dấu, hoạt động ở chế độ không đồng bộ hoặc ở chế độ đồng bộ với xung đồng hồ.



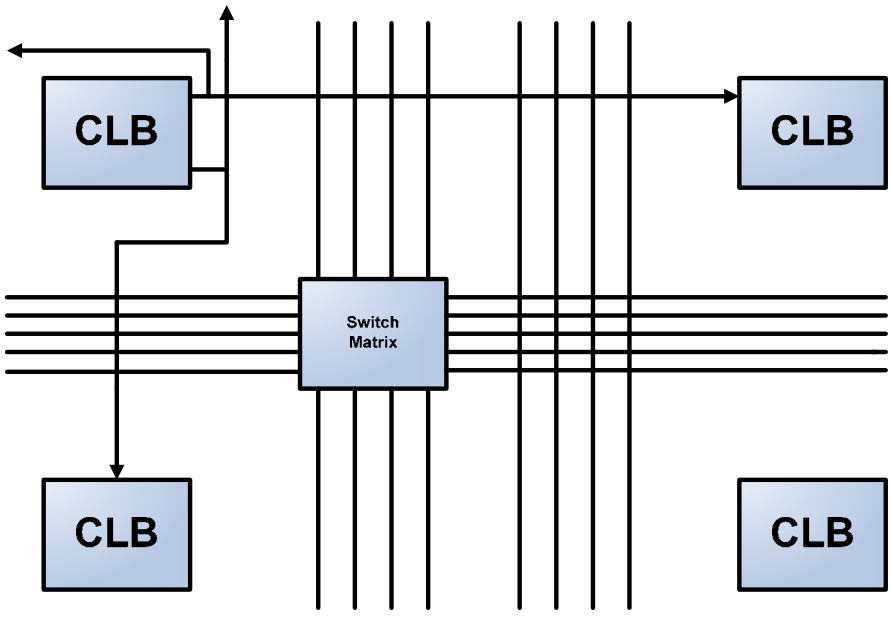
Hình 2.6: Bộ nhân không đồng bộ và đồng bộ của Xilinx

Tương tự như bộ nhân, bộ cộng cũng được nhúng vào trong chip FPGA. Ngoài ra, còn một phép toán cũng rất thường xuyên được sử dụng trong các ứng dụng DSP là phép toán nhân cộng tích lũy MAC. Thông thường để thực hiện phép toán này sẽ phải kết hợp các phép nhân, cộng, các CLB và cả các Flip-Flop, sau đó kết quả cần được lưu vào RAM. Do đó trong một số chip FPGA cũng đã được

nhúng thêm các bộ MAC, điều này giúp cho công việc của kỹ sư thiết kế được đơn giản hơn rất nhiều.

Liên kết khả trình

Liên kết ở FPGA khác xa so với ở CPLD, tuy nhiên lại giống với của dãy cổng ASIC. Có một line dài được dùng để nối các CLBs quan trọng mà chúng lại ở cách xa nhau mà không gây ra quá nhiều trễ. Chúng có thể được dùng như là các bus ở trong chip. Có các line ngắn được dùng để liên kết các CLBs riêng rẽ nhưng đặt gần nhau. Và cũng thường có vài ma trận chuyển mạch, giống như trong CPLD, nối giữa các line dài và ngắn lại với nhau theo một số cách đặc biệt. Các chuyển mạch có thể lập trình được bên trong chip cho phép kết nối giữa CLBs tới các đường kết nối và giữa đường kết nối với các line khác và với ma trận chuyển mạch. Các bộ đệm ba trạng thái được dùng để kết nối phần lớn các CLBs với các đường kết nối dài, tạo nên các Bus.



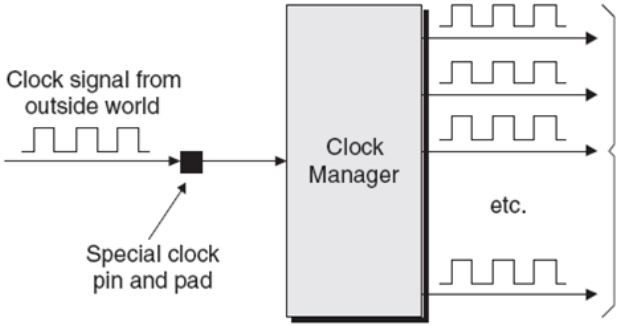
Hình 2.7: Liên kết khả trình

Khối quản lí xung đồng hồ DCM

Tất cả các thành phần đồng bộ trong chip FPGA đều phải được điều khiển bởi tín hiệu đồng hồ clock. Về cơ bản các tín hiệu đồng hồ này được đưa từ bên ngoài vào chip FPGA thông qua một chân riêng là chân clock và sau đó được đưa vào các kết nối để đến các thành phần tương ứng.

Trong các chip FPGA có một hệ thống các kết nối riêng để đưa tín hiệu clock nối đến các thành phần tương ứng. Kiến trúc này đảm bảo các thành phần ở các vị trí khác nhau sẽ nhận được tín hiệu xung đồng hồ có đặc tính giống nhau, tránh trường hợp bị trượt xung đồng hồ do quá trình truyền tín hiệu với các khoảng cách khác nhau. Thông thường một chip FPGA sẽ có nhiều chân clock và nhiều hệ thống kết nối khác nhau đảm bảo chip FPGA có thể hoạt động ở nhiều tần số khác nhau.

Trên thực tế thay vì kết nối trực tiếp, các chân clock thường được nối qua một khối phần cứng điều khiển đồng hồ số DCM. Các clock được tạo ra từ khối DCM sau đó được đưa đến hệ thống kết nối hoặc đưa ra ngoài để điều khiển các thiết bị ngoại vi.



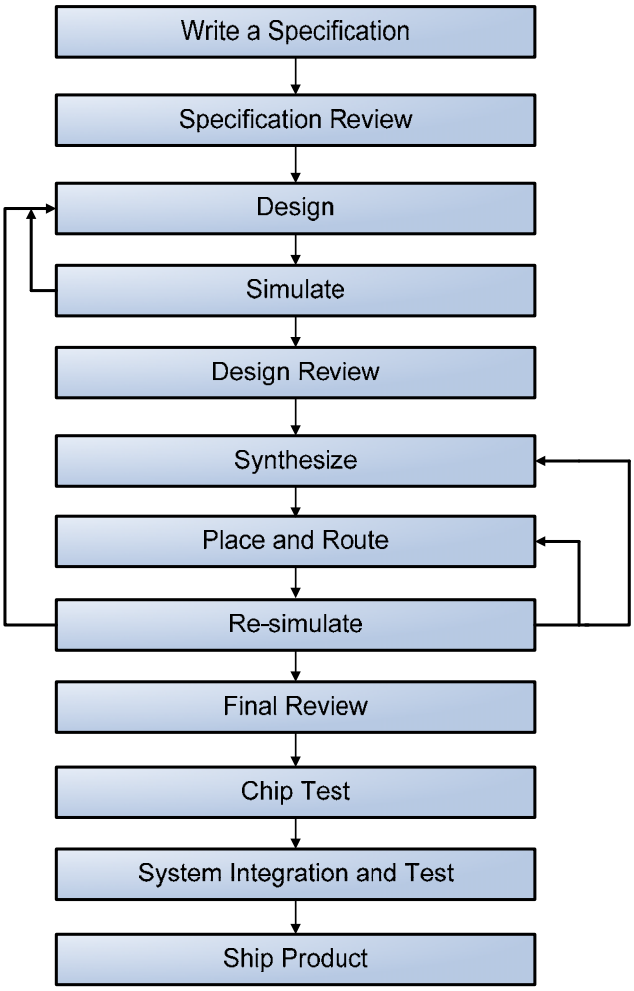
Hình 2.8: Khối DCM

Mỗi dòng chip FPGA khác nhau có thể có các khối DCM khác nhau và có nhiều khối DCM trong một chip. Về cơ bản, một khối DCM thực hiện các chức năng sau:

- Khử nhiễu
- Tổng hợp tần số
- Dịch pha
- Tự điều chỉnh trượt đồng hồ

2.1.3. Trình tự thiết kế một chip

Trình tự thiết kế một hệ thống trên nền FPGA bao gồm các bước sau [13]:



Hình 2.9: Tiến trình thiết kế

Đưa ra đặc tả kĩ thuật

Tầm quan trọng của các đặc tả kĩ thuật không thể phóng đại quá. Nó chỉ tuyệt đối cần thiết khi là một hướng dẫn để chọn công nghệ phù hợp và tạo những yêu cầu của người thiết kế cho các nhà sản xuất chip. Các đặc tả kĩ thuật cho phép mỗi kĩ sư hiểu về thiết kế hệ thống chung và công việc của họ trong hệ thống đó là gì, thiết kế giao diện đúng cho một loạt các phần của chip. Các đặc tả kĩ thuật cũng giúp tiết kiệm thời gian và sự hiểu lầm.

Chi tiết kĩ thuật nên bao gồm các thông tin sau đây:

- Sơ đồ khối bên ngoài để chỉ ra vị trí của chip trong hệ thống.
- Sơ đồ khối bên trong chỉ rõ mỗi chức năng của các thành phần.
- Miêu tả các chân vào ra bao gồm khả năng lái đầu ra, mức ngưỡng đầu vào.
- Thời gian ước lượng bao gồm thời gian thiết lập và giữ ở các chân vào, thời gian lan truyền ra các cổng ra và thời gian chu kì đồng hồ.
- Đếm xấp xỉ số gate.
- Dạng đóng gói.
- Tiêu thụ nguồn.
- Giá cả.
- Các thủ tục để kiểm tra.

Xem xét đặc tả kĩ thuật

Mỗi khi một chi tiết miêu tả kĩ thuật được xuất bản, nó có thể được dùng để chọn nhà sản xuất chip tốt nhất với công nghệ và cấu trúc giá cả là tốt nhất đáp ứng được yêu cầu của bạn.

- Chọn một hướng tiếp cận thiết kế: Tại thời điểm này bạn phải quyết định cách thực hiện thiết kế mà bạn mong muốn. Đối với các chip nhỏ thì cách tiếp cận bằng sơ đồ nguyên lý thường được chọn, đặc biệt là khi các kĩ sư thiết kế đã quen thuộc với các công cụ này. Thế nhưng đối với các thiết kế lớn hơn, ngôn ngữ miêu tả phần cứng HDL như Verilog và VHDL được dùng bởi khả năng mềm dẻo, dễ đọc, dễ chuyển giao. Khi dùng ngôn ngữ cấp cao, phần mềm tổng hợp sẽ được yêu cầu tổng hợp thiết kế, các phần mềm này sẽ tạo ra các cổng ở cấp thấp từ miêu tả ở cấp cao hơn.
- Chọn công cụ tổng hợp: Tại điểm này, bạn phải quyết định chọn phần mềm tổng hợp nào sẽ được dùng nếu bạn có kế hoạch thiết kế FPGA với HDL. Điều đó rất quan trọng kể từ khi mỗi công cụ tổng hợp được khuyên dùng và sự ủy thác của cách thiết kế phần cứng nên nó có thể hoạt động tổng hợp đúng hơn.

Thiết kế chip

Có một số cách để thiết kế chip:

- Top-down design
- Macros
- Synchronous design
- Protect against metastability
- Avoid floating nodes
- Avoid bus contention

Mô phỏng

Mô phỏng là một quá trình được thực hiện sau khi thiết kế xong. Từng phần nhỏ của thiết kế nên được mô phỏng trước khi kết hợp chúng thành các phần lớn hơn. Điều này rất là cần thiết và sự mô phỏng theo thứ tự sẽ kiểm tra chức năng hoạt động đúng của từng phần. Mỗi khi thiết kế và mô phỏng hoàn thành, dẫn đến một cái nhìn tổng quan khác về thiết kế vì thế thiết kế có thể được kiểm tra lại. Mô phỏng đúng và thành công thì bạn sẽ biết được chip của bạn sẽ hoạt động đúng trong hệ thống.

Tổng hợp

Nếu thiết kế dùng HDL, bước tiếp theo là tổng hợp chip, bao gồm việc dùng phần mềm tổng hợp để chuyển đổi thật tối ưu từ thiết kế mức RTL sang thiết kế mức gate mà có thể gắn vào các khối logic trong FPGA.

Sắp đặt chip

Bước tiếp theo là sắp đặt chip, kết quả trong việc thiết kế vật lý cho chip thực. Điều này bao gồm các công cụ của nhà sản xuất để tối ưu lập trình cho chip để thực hiện thiết kế. Sau đó, thiết kế được lập trình vào cho chip.

Mô phỏng lại

Sau khi sắp đặt xong, thì chip phải được mô phỏng lại với các con số về thời gian tạo ra bởi các layout thực tế. Nếu mọi thứ đều tốt đến thời điểm này, thì một kết quả mô phỏng mới sẽ đúng với các kết quả dự đoán.

Kiểm tra chip

Đối với các thiết bị lập trình được, đơn giản là lập trình thiết bị đó và ngay lập tức có mẫu thử, sau đó đặt mẫu thử này vào trong hệ thống và xem hệ thống có làm việc đúng không. Nếu làm lần lượt các bước ở trên thì đa phần là hệ thống sẽ hoạt động đúng chỉ với một vài lỗi rất nhỏ. Các lỗi này cần được kiểm tra và trích dẫn lại để có thể được sửa chữa trong phiên bản tiếp theo của chip. Trước khi các chip được đưa vào sản xuất, rất cần thiết có một vài kiểm tra hệ thống qua thời gian dài. Nếu một chip được thiết kế đúng, thì nó chỉ bị lỗi điện học hoặc lỗi cơ học – những lỗi này sẽ thường xuyên xảy ra với loại kiểm tra khắc nghiệt này.

2.1.4. Giới thiệu ngôn ngữ HDL

HDL (Hardware Description Language) là ngôn ngữ mô tả phần cứng, mô tả hành vi của mạch điện hoặc hệ thống, từ đó mạch điện vật lý hoặc hệ thống có thể được thực thi.

Động cơ thúc đẩy cơ bản khi dùng ngôn ngữ HDL do đây là một ngôn ngữ chuẩn của các nhà công nghệ, các nhà phân phối do đó chúng có khả năng tùy biến và kế thừa cao. Hai ứng dụng trực tiếp chính của HDL là trong mảng các thiết bị logic lập trình được là CPLDs và FPGAs. Mỗi khi mã nguồn HDL được viết, chúng có thể được dùng để thực thi mạch điện trong các thiết bị lập trình được (từ Altera, Xilinx, Almel, ...) hoặc có thể gửi đến các xưởng chế tạo các chip ASIC. Hiện này, rất nhiều các chip thương mại phức tạp ví dụ như các vi điều khiển được thiết kế dựa trên cách tiếp cận này. Một điều chú ý là HDL là trái ngược với các chương trình máy tính thông thường, câu lệnh được thực hiện tuần tự thì các câu lệnh HDL được thực hiện song song. Vì lí do đó, nên HDL thường được coi là một

mã nguồn hơn là một chương trình. Trong HDL chỉ có một vài trường hợp câu lệnh được thực thi tuần tự.

Có hai ngôn ngữ HDL được chuẩn công nghiệp qui định là VerilogHDL và VHDL. Hiện nay vẫn còn nhiều sự tranh cãi xung quanh việc ngôn ngữ nào mạnh hơn, tốt hơn, tuy nhiên, người lập trình có thể chọn ngôn ngữ lập trình dựa trên sự quen thuộc với ngôn ngữ đó và thấy dễ dàng khi triển khai hệ thống. Chúng ta có thể dễ dàng chuyển từ VHDL sang Verilog và ngược lại.

VHDL

VHDL là viết tắt của VHSIC Hardware Description Language. Bản thân VHSIC là viết tắt của Very High Speed Integrated Circuits (mạch tích hợp tốc độ cao), lần đầu tiên được sáng lập bởi United State Department of Defense trong những năm 80. Phiên bản đầu tiên là VHDL 87, lần nâng cấp sau đó có tên là VHDL 93. VHDL là ngôn ngữ mô tả phần cứng nguyên gốc đầu tiên được chuẩn hóa bởi Institue of Electrical and Electronics Engineers (IEEE), tới chuẩn IEEE 1076. Trong IEEE 1164, có một chuẩn được thêm vào là giới thiệu hệ thống logic đa giá trị (multi-valued logic system).

Verilog HDL

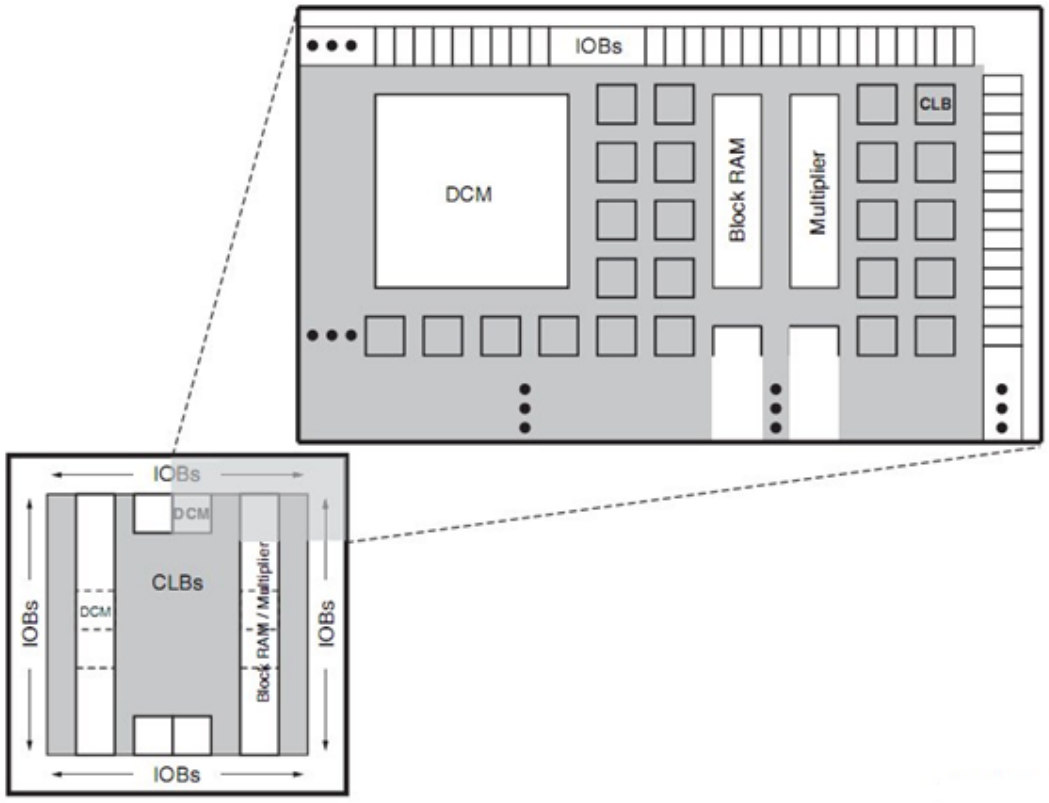
Verilog HDL được phát minh bởi Phil Moorby và Prabhu Goel khoảng mùa đông năm 1983/1984 tại Automated Integrated Design Systems (sau đổi tên thành Gateway Design Automation vào năm 1985). Do có cấu trúc tương tự với C nên Verilog rất gần gũi với kĩ sư và được sử dụng là ngôn ngữ lập trình trong nhiều ứng dụng. Verilog có nhiều chuẩn: Verilog 97, Verilog 2001 và Verilog 2005. Trong ngôn ngữ Verilog, các lệnh được thực hiện song song, chỉ khi các câu lệnh đặt trong *always* hay *initial* thì các lệnh trong đó mới được thực hiện tuần tự.

2.2. Giới thiệu cấu trúc FPGA của Xilinx Spartan-3

Cấu trúc tổng quan của Spartan-3 gồm có năm thành phần có chức năng khả trình cơ bản sau:

- CLBs bao gồm các LUTs rất linh động có chức năng thực thi các logic và các phần tử nhớ dùng như là các Flip-Flop hoặc các chốt. CLB thực hiện phần lớn các chức năng logic như là lưu dữ liệu, …
- IOBs điều khiển dòng dữ liệu dữa các chân vào ra I/O và các logic bên trong của FPGA. IOBs hỗ trợ luồng dữ liệu hai chiều và hoạt động logic ba trạng thái. Hỗ trợ phần lớn các chuẩn tín hiệu, bao gồm một vài chuẩn tốc độ cao như DDR.
- Block RAM cho phép lưu trữ dữ liệu dưới dạng các khối dual-port 18-Kbit.
- Multiplier Blocks cho phép hai số nhị phân 18 bits làm đầu vào và dễ dàng tính toán tích của chúng.
- Khối DCM cung cấp khả năng tự xác định xung clock, là giải pháp số hoàn chỉnh cho các tín hiệu clock phân phối, trễ, nhân, chia và dịch bit.

Các phần tử này được tổ chức như trong hình sau:



Hình 2.10: Cấu trúc các thành phần của Spartan-3

Từ hình vẽ thấy rằng, các IOBs bao quanh các mảng CLBs, riêng Spartan-3E chỉ có một vòng các IOBs. Mỗi cột block RAM bao gồm một vài block RAM 18-Kbit, mỗi block RAM lại gắn liền với một bộ Multiplier dành riêng. Các DCM được đặt ở các vị trí: hai DCM phía trên và hai cái phía dưới của thiết bị, và đối với các thiết bị lớn hơn thì có thêm các DCM ở phía bên cạnh [7]

Đặc điểm chung mạng Spartan-3 là kết nối liên thông giữa năm phần tử cơ bản này và truyền tín hiệu giữa chúng. Mỗi thành phần chức năng này có một ma trận chuyển mạch dành riêng để cho phép chọn lựa kết nối cho phép chọn lựa kết nối cho việc đi dây trong FPGA.

2.3. Triển khai phần cứng cho mạng nơ-ron trên nền FPGA

Để thực hiện triển khai phần cứng mạng nơ-ron trên nền FPGA sử dụng ngôn ngữ VHDL là một ngôn ngữ mô tả phần cứng làm đơn giản hóa sự phát triển các hệ thống phức tạp. Bởi vì nó có thể mô hình hóa và mô phỏng một hệ thống số có mức trừu tượng cao và với các tiện ích quan trọng cho các thiết kế mô-đun. Trong phần này sẽ giới thiệu thiết kế các mô hình nơ-ron nhân tạo dựa trên thiết bị FPGA Xilinx, những tính năng mô tả dưới đây cần thiết cho việc thực thi nơ-ron:

-Logic nhanh cho phép việc thiết kế các hàm số học nhanh và gọn (ví dụ: Phép nhân và phép cộng)

-Các bảng tìm kiếm LUT có thể được sử dụng như các RAMs và ROMs

-Các hàm kết hợp có tới mười đầu vào ở trong các khối logic khả cấu hình CLBs

-Khả năng định tuyến rất cao cho phép triển khai thành công các trễ path quan trọng, thậm chí cho mạng nơ-ron phức tạp.

Các mô-đun khác nhau trong việc triển khai nơ-ron nhân tạo là bộ cộng, bộ nhân và bộ sinh hàm Sigmoid. Có rất nhiều phương pháp hữu hiệu để triển khai hàm Sigmoid.

2.4. Các phương pháp khác nhau triển khai hàm kích hoạt

2.4.1. Bảng tìm kiếm LUT

Hàm Sigmoid được thực hiện khó khăn trên một nền thiếu khối dấu phẩy động. Để giảm các chi phí thực hiện, hàm truyền Sigmoid được thay thế bởi một bảng LUT tính toán từng bước cố định ít tốn kém hơn và một phép nội suy tuyến tính được thực hiện để thay thế hàm Sigmoid. Một LUT có thể được sử dụng để thực hiện hàm kích hoạt Sigmoid bởi các giá trị trung bình rời rạc. Nhưng nó tiêu thụ một phạm vi lưu trữ lớn khi ở mức độ chính xác cao. Nếu dải đầu vào cần 21 bits và độ chính xác cần 16 bits để trình bày kết quả của LUT, thì cần 4MB LUT. Nó tiêu thụ một phạm vi và thời gian lớn, có thể ảnh hưởng đến tốc độ tính toán. Nếu chuỗi Taylor được dùng đển tính hàm số mũ, việc tính toán bao gồm nhiều bộ nhân và vì vậy làm tăng diện tích phần cứng. Thiết kế LUT không tối ưu tốt dưới dạng FLP (Floating Point) và do đó định dạng dấu phẩy tĩnh được sử dụng. Nhưng việc thực hiện hàm Log-Sigmoid trên chip làm tăng kích cỡ phần cứng một cách đáng kể. Để tối ưu hóa diện tích để mở rộng, RAM bên trong có sẵn trên FPGAs được sử dụng để thực hiện LUT làm cơ sở cho hàm kích hoạt. Nó làm giảm diện tích và tăng tốc độ. Nếu độ chính xác được cải thiện, thì phần cứng được thực thi cho việc xấp xỉ PWL (Piecewise Linear) tuyến tính từng khúc, phương pháp triển khai hàm kích hoạt được sử dụng.

2.4.2. Xấp xỉ tuyến tính

Việc xấp xỉ này thực hiện đơn giản và được đưa ra như sau:

$$f(x) = \begin{cases} 1 \text{ nếu } x \geq h \\ \frac{1}{2}(1 + \frac{x}{h}) \text{nếu } |x| \leq h \\ 0 \text{ nếu } x \leq -h \end{cases}$$

Trong đó h là một tham số tùy chỉnh. Đây là một hàm tuyến tính trong miền [-h,h] với độ dốc là h/2 [23]

2.4.3. Xấp xỉ đa thức

Phương pháp này gồm có việc sử dụng một đa thức P, bậc N, đưa ra việc xấp xỉ tương ứng cho hàm Sigmoid trên một khoảng I = [a,b]. Tồn tại nhiều phương pháp khác nhau để triển khai việc xấp xỉ này, ví dụ một trong các phương pháp dựa trên cơ sở chuỗi Taylor tập trung quanh điểm trung tâm của khoảng đó. Tuy nhiên, điều này làm cho viêc xấp xỉ tốt hơn chỉ xung quanh điểm quan tâm và không phải trên toàn bộ khoảng đó. Một xấp xỉ đa thức bậc năm của hàm Sigmoid trong miền I = [0,5] được thực hiện như sau:

$P(x) = \alpha + (\beta + (\gamma + (\delta + (\varepsilon + \theta.x).x).x).x).x$ với:

α = 0.49999351; β = 0.25276133

γ = – 0.00468879; δ = – 0.02246096

ε = 0.00541780

θ = – 0.00039438

Phương pháp này có sai lệch tương đối là 0.689 x 10^{-3}

2.4.4. Xấp xỉ bậc hai

Chú ý tới phương trình bậc hai:

$y = ax^2 + bx + c$

Sử dụng phương pháp xấp xỉ bình phương tối thiểu với x Є [0;4] được:

$y = -0.0363x^2 + 0.2610x + 0.5028$

Hàm này có thể được viết đơn giản hơn là:

$y = 0.972 - 0.57(0.25x - 0.898)^2$

Kiểm nghiệm với hàm Sigmoid nguyên bản, người ta thấy rằng đây là một hàm không đối xứng qua $y = 0.5$. Thêm vào đó, hàm Sigmoid có hai đường tiệm cận với $y = 0$ và $y = 1$. Do vậy việc đánh giá hàm cho $x < 0$ là phần bù bổ sung với $x > 0$. Hàm xấp xỉ có thể được biểu diễn như sau:

$$y = \begin{cases} 0.972 \quad 0.57(0.25x \quad 0.898)^2 \; với \; 0 < x < 4 \\ 0.028 + 0.57(0.25|x| - 0.898)^2 \; với \; -4 < x < 0 \end{cases}$$

CHƯƠNG 3: THUẬT TOÁN CORDIC

Thuật toán CORDIC (COrdinate Rotation Digital Computer) là một kĩ thuật được sử dụng để tính giá trị của các hàm lượng giác. CORDIC khác với các phương pháp tính toán khác bởi vì nó có thể dễ dàng được thực hiện chỉ bằng việc sử dụng các phép toán cộng, trừ và dịch bit. Nó không nhanh như phương pháp sử dụng bảng LUT nhưng phương pháp dùng bảng LUT cần nhiều bộ nhớ cho các trọng số weight, do vậy phương pháp CORDIC có thể sử dụng vùng chip ít hơn một cách đáng kể, điều này thiết thực cho các những ứng dụng mà diện tích quan trọng hơn tính năng kỹ thuật [7].

So với phương pháp xấp xỉ tuyến tính, phương pháp sử dụng thuật toán CORDIC có độ chính xác cao hơn. Do phương pháp xấp xỉ này thực hiện xấp xỉ hàm đại số phức tạp bằng các hàm bậc nhất trên các đoạn khác nhau. Nên đầu ra trong phương pháp xấp xỉ tuyến tính gấp khúc phân mảnh. Độ chính xác của kỹ thuật này phụ thuộc số đoạn chia và cách thức chia đoạn [23]

Còn phương pháp xấp xỉ bậc hai thì cần tới ba bộ nhân, chi phí và giá thành sẽ cao khi thực hiện triển khai phần cứng.

Chính vì những lý do trên nên trong nội dung luận văn này, thuật toán CORDIC là một giải pháp rất phù hợp được học viên lựa chọn để thực hiện.

3.1. Giới thiệu về thuật toán CORDIC

Thuật toán CORDIC được đưa ra đầu tiên bởi Volder trong bài báo "The CORDIC Trigonometric Computing Technique", xuất bản năm 1959 [22]. Ban đầu người ta dự định sử dụng nó cho sự tính toán thời gian thực trên máy bay, nhưng sau đó đã tìm ra nhiều ứng dụng khác nữa. Năm 1971, Walther đã chứng minh được rằng phạm vi của thuật toán cũng có thể được mở rộng ra tính toán các phương trình lượng giác với cả các hàm hyperbolic và tuyến tính (nhân-cộng-hỗn hợp).

Walther đã trình bày việc thực hiện triển khai phần cứng sử dụng ba thanh ghi n-bit, ba bộ cộng/ trừ n-bit, ba bộ dịch và một bảng LUT nhỏ. Volder đã trình bày một thiết kế nối tiếp gồm có ba thanh ghi n-bit, ba bộ cộng/ trừ 1-bit, và một số

cổng dịch được chỉ định các bits từ các thanh ghi để đưa vào trong các bộ cộng/ bộ trừ. Luận văn này sẽ thực hiện cả việc triển khai và phân tích năng lực thuật toán được sử dụng trong mạng Neural Network.

3.2. Mô tả thuật toán

Thuật toán thực hiện một trong hai kiểu: Rotation (Quay) hoặc Vectoring (Véc-tơ). Hai kiểu xác định tập các hàm có thể được tính toán sử dụng trong thuật toán. Trong kiểu Rotation, thành phần x- và y- của véc-tơ khởi tạo là đầu vào, cũng như là một góc quay. Sau đó phần cứng tính toán lặp đi lặp lại các thành phần x- và y- của véc-tơ sau khi nó được quay bởi các góc quay lý thuyết [8]

Trong kiểu Vectoring, hai thành phần đầu vào, biên độ và góc của vectơ gốc được tính toán. Điều này được thực hiện bằng cách quay vectơ đầu vào cho đến khi nó được sắp thẳng hàng với trục x-. Bằng cách ghi lại các góc quay để đạt được sự sắp xếp này, khi biết trước góc của vectơ gốc. Khi thuật toán hoàn thành, thành phần x- của vectơ sẽ bằng với biên độ của vectơ khởi đầu.

Thuật toán CORDIC thực hiện việc tính toán của nó chỉ sử dụng các phép toán công, trừ, và dịch. Điều này được thực hiện bằng việc quay lặp đi lặp lại vectơ đầu vào và đồng qui một cách chậm chạp tới vectơ quay cuối cùng. Điều này được thực hiện trong một chuỗi các bước hoàn toàn xác định. Bước quay đầu tiên nhìn vectơ quay $\frac{\pi}{4}$ radians:

(3.1) $y_1 = \pm x_0 = R_0 \sin(\theta_0 \pm \frac{\pi}{4})$

$x_1 = \mp y_0 = R_0 \cos(\theta_0 \pm \frac{\pi}{4})$

Trong đó x_0 và y_0 là các vectơ đầu vào được gắn ở gốc với biên độ R_0 và góc θ_0 :

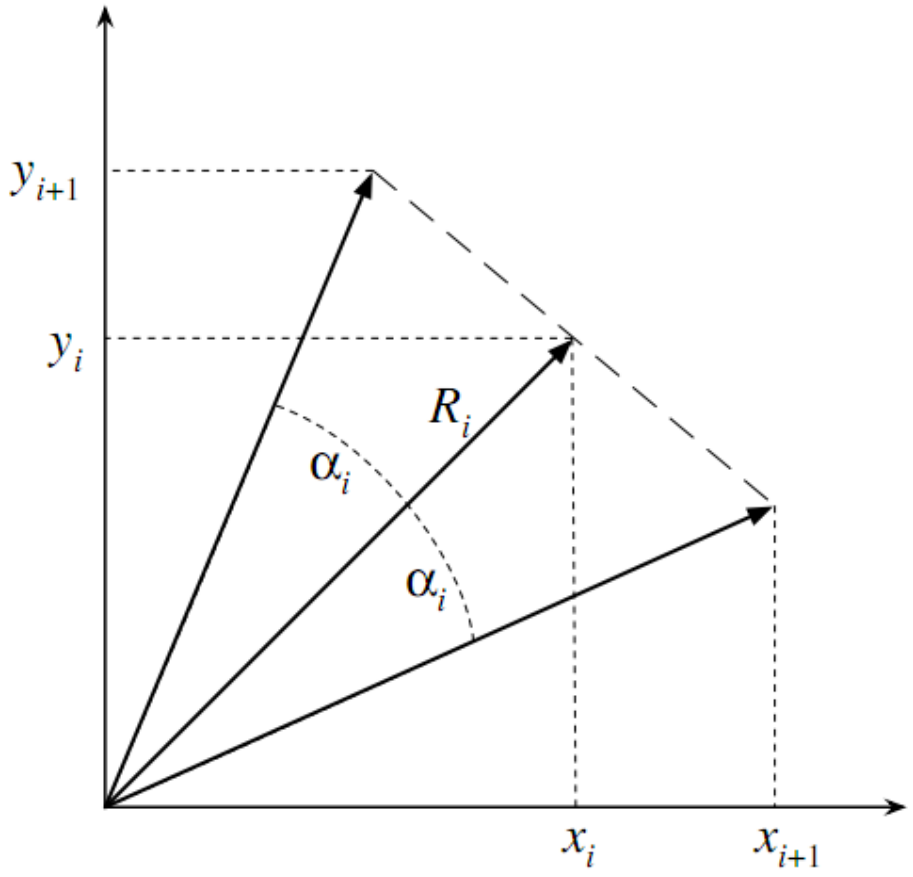
(3.2) $x_i = R_0 \cos \theta_i$

$y_i = R_0 \sin \theta_i$

Trong mỗi bước xảy ra sau, một góc quay mới α_i được xác định như sau:

(3.3) $\alpha_i = tan^{-1}(2^{-1})$, trong đó i > 0

Sự giới hạn được quyết định trong sự cho phép các phép tính quay biểu diễn trong mỗi bước được thực hiện chỉ sử dụng một phép cộng (hoặc trừ) và một phép dịch.



Hình 3.1: Bước thứ i trong thuật toán CORDIC

Hình 3.1 biểu diễn mỗi bước trong thuật toán CORDIC trông như thế nào. Tại mỗi bước, một quyết định được đưa ra để thực hiện quay vectơ theo góc hoặc là $+\alpha_i$ hoặc là $-\alpha_i$. Kết quả của cả hai quyết định này được nhìn thấy ở hình trên. Biểu thức để quay vectơ trong bước thứ (i+1) là:

(3.4) $x_{i+1} = \sqrt{1 + 2^{-2i}} \cos(\theta_i + \alpha_i)$

$y_{i+1} = \sqrt{1 + 2^{-2i}} \sin(\theta_i + \alpha_i)$

Khi đặt giới hạn trong phương trình (4.3), quá trình dịch và cộng trở thành:

(3.5) $x_{i+1}=\frac{1}{K_i}(x_i - d_i 2^{-i} y_i)$

$y_{i+1}=\frac{1}{K_i}(y_i + d_i 2^{-i} x_i)$

Trong đó, $K_i=\sqrt{1 + 2^{-2i}}$, $d_i=\pm 1$

K_i là thành phần giới hạn lỗi biên độ và d_i tương ứng với hướng quay (+1 tương ứng với việc quay từ trục x-). Các thành phần 2^{-i} trong các phương trình trên và dưới tương ứng lần lượt là sự dịch trái của y_i và x_i khi tính toán theo cơ số hai. Giá trị dịch này sau đó được cộng (hoặc trừ) cho giá trị hiện tại của thành phần. Volder gọi phép tính này là *cross-addition.*Nó cho phép thuật toán được sử dụng một cách hiệu quả trong phần cứng số.

Như mô tả trong hình 3.1, Tất cả các bước quay ảnh hưởng đến việc tăng độ lớn biên độ của véc-tơ đầu vào bởi hệ số $\sqrt{1 - 2^{-2i}}$ với mỗi lần quay. Sai số này được đưa ra như là kết quả của phép tính thu được của thuật toán từ biến đổi Givens, quay một véc-tơ bởi một góc lý thuyết đã định rõ:

(3.6) $x'= x\cos\phi - y\sin\phi$

$y'= y\cos\phi - x\sin\phi$

Các chỉ số x’ và y’ có thể được tính lại bằng việc sử dụng công thức lượng giác cơ bản $\tan\alpha=\frac{\sin\alpha}{\cos\alpha}$ cho ra kết quả sau:

(3.7) $x'= \cos\phi(x - y\tan\phi)$

$y'= \cos\phi(y - x\tan\phi)$

Cũng sử dụng giới hạn trong phương trình (3.3), chúng ta cũng có được các mối liên hệ như trong phương trình (3.5). Giới hạn sai số từ sự hiện diện của hệ số cosine trong phương trình (3.7), độc lập với hướng quay. Sine cosine đối xứng với nhau về hướng quay, sine cosine đối xứng qua trục y-. Thêm vào đó, với việc tích lũy sai số của mỗi bước quay, do đó nếu số lần lặp được thiết lập, sai số tổng cộng trong thuật tóan độc lập với góc quay đầu vào.

Nếu lần quay đầu tiên là:

(3.8) $x_1 = \sqrt{1 + 2^{-2(1)}}\, R_0 \cos(\theta_0 + d_0\alpha_0)$

$$y_1 = \sqrt{1 + 2^{-2(1)}}\, R_0 \sin(\theta_0 + d_0\alpha_0)$$

Sau đó, lần quay thứ hai được tính như sau:

(3.9) $$x_2 = \sqrt{1 + 2^{-2(1)}}\sqrt{1 + 2^{-2(2)}}\, R_0 \cos(\theta_0 + d_0\alpha_0 + d_1\alpha_1)$$

$$y_2 = \sqrt{1 + 2^{-2(1)}}\sqrt{1 + 2^{-2(2)}}\, R_0 \sin(\theta_0 + d_0\alpha_0 + d_1\alpha_1)$$

Suy rộng ra, lần quay thứ n sẽ được tính như sau và dẫn đến công thức định nghĩa tổng quát:

(3.10) $$x_{n+1} = [\sqrt{1 + 2^{-2(1)}}\sqrt{1 + 2^{-2(2)}} \ldots \sqrt{1 + 2^{-2(n)}}]$$

$$R_0 \cos(\theta_0 + d_0\alpha_0 + d_1\alpha_1 + \cdots + d_n\alpha_n)$$

$$y_{n+1} = [\sqrt{1 + 2^{-2(1)}}\sqrt{1 + 2^{-2(2)}} \ldots \sqrt{1 + 2^{-2(n)}}]$$

$$R_0 \sin(\theta_0 + d_0\alpha_0 + d_1\alpha_1 + \cdots + d_n\alpha_n)$$

Sự gia tăng độ lớn biên độ tổng có thể được định ra bằng hệ số $K_n = \prod_n \sqrt{1 + 2^{-2k}}$. Sự gia tăng này phải được tính toán khi các phép tính thực hiện sử dụng thuật toán này.

3.3. Các thanh ghi tích lũy

Thiết kế CORDIC sử dụng đến ba thanh ghi tích lũy: Thanh ghi X, thanh ghi Y và bộ tích lũy góc Z. Các thanh ghi X và Y để lưu giữ giá trị thành phần véc-tơ trục x- và trục y- hiện tại khi nó đang được thực hiện quay. Bộ tích lũy góc lưu giữ số lượng quay tổng cộng đã hoàn thành ở lần lặp hiện thời.

Bộ tích lũy góc lưu giữ các đối số thành phần sine và cosine tính được ở phương trình (3.10): $Z_n = d_0\alpha_0 + d_1\alpha_1 + \cdots + d_n\alpha_n$. Tuy nhiên, do số hạng này tương đương với góc quay mong muốn – tính thay thế được $\alpha_i = \tan^{-1}(2^{-i})$ - thanh ghi Z phải luôn được tính theo biểu thức:

(3.11) $$Z_i = \sum_{n-1}^{i} d_n \tan^{-1}(2^{-i})$$

Các số hạng đối số có thể được lưu giữ trong một bảng LUT nhỏ. Khi điều này được hoàn thành xong, thì chỉ có một trong hai phép cộng hoặc phép trừ cần dùng đến để tính giá trị tiếp theo trong thanh ghi Z, do d= ±1. Bảng này rất nhỏ, chỉ cần

một hàng cho mỗi lần lặp của thuật toán. Do đó chỉ số đếm lặp có thể là cố định không đổi trên phần cứng, nên kích cỡ của bảng là hằng số.

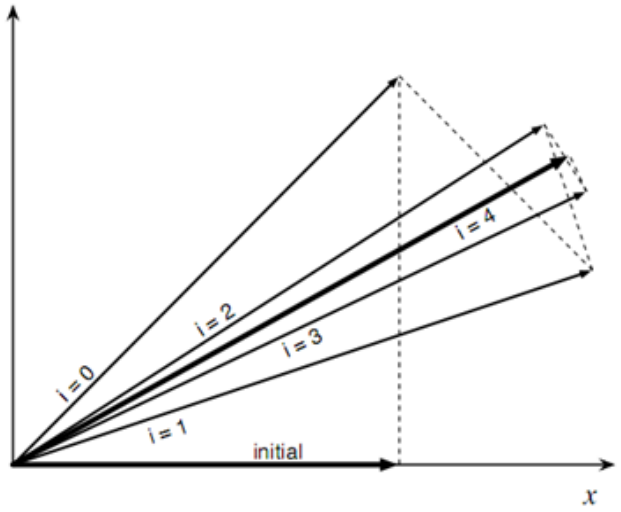
Các thanh ghi tích lũy cũng được sử dụng để xác định hướng quay. Trong kiểu quay Rotation Mode, dấu của thanh ghi Z xác định hướng. Trong kiểu Vectoring Mode, thanh ghi Y xác định hướng. Các kiểu này sẽ được tìm hiểu chi tiết hơn trong phần sau.

3.4. Các kiểu tính toán

Thuật toán CORDIC thực hiện bởi một trong hai kiểu: Rotation và Vectoring. Mô hình thực hiện xác định tập hợp các hàm có thể được tính toán và các giá trị trong những thanh ghi X, Y và Z thay đổi ở mỗi lần lặp như thế nào.

3.4.1. Rotation Mode

i	*X*	*Y*	*Z*
—	**1.0000**	**0.0000**	**30.0000°**
0	1.0000	1.0000	–15.0000°
1	1.5000	0.5000	11.5651°
2	1.3750	0.8750	–2.4712°
3	1.4844	0.7031	4.6538°
4	1.4404	0.7959	1.0775°
5	1.4156	0.8409	–0.7124°
6	1.4287	0.8188	0.1828°
7	1.4223	0.8300	–0.2649°
8	1.4255	0.8244	–0.0411°
9	1.4272	0.8216	0.0709°
10	**1.4264**	**0.8230**	**0.0149°**



Hình 3.2: Chế độ Rotation của thuật toán CORDIC

Trong Rotation Mode, véc-tơ đầu vào được quay bởi một góc lý thuyết định trước. Véc-tơ đầu vào được định rõ là giá trị khởi tạo các thanh ghi X và thanh ghi Y. Số lần quay là đầu vào ở trong thanh ghi Z. Để quay véc-tơ đầu vào qua góc đầu vào, đích của mỗi lần lặp sẽ bị làm giảm giá trị trong bộ tích lũy góc đến 0. Do các giá trị arctangent cho mỗi lần lặp là cố định, chỉ có giá trị trong thanh ghi Z là có thể

được điều khiển thông qua các giá trị d. Trong Rotation Mode, Các giá trị quyết định được định nghĩa:

$$d_i = \begin{cases} +1, \text{nếu } Z_i \geq 0 \\ -1, \text{nếu } Z_i < 0 \end{cases} \quad (3.12)$$

Với định nghĩa hàm quyết định này, sau n lần lặp của thuật toán, các giá trị trong mỗi thanh ghi sẽ là:

$$\begin{aligned} X_n &= K_n[X_0 \cos(Z_0) - Y_0 \sin(Z_0)] \\ Y_n &= K_n[Y_0 \cos(Z_0) + X_0 \sin(Z_0)] \quad (3.13) \\ Z_n &= 0 \\ K_n &= \prod_n \sqrt{1 + 2^{-2n}} \end{aligned}$$

Hình 3.2 cho thấy điều xảy ra với véc-tơ đầu vào sau mỗi lần lặp của thuật toán. Trong ví dụ này, véc-tơ bắt đầu được gán với trục x-. Độ lớn biên độ của véc-tơ tăng lên qua mỗi bước, khi góc của véc-tơ hội tụ ở 30^0. Các giá trị của các thanh ghi X, Y và Z cũng được thể hiện.

Trong các phần tiếp theo sẽ mô tả các hàm khác nhau được tính toán như thế nào khi sử dụng Rotation Mode.

Sine, Cosine và phép biến đổi Polar-Cartesian

Việc tính toán các hàm sine và cosine là thực chất của Rotation Mode, và có thể dễ dàng thu được từ phương trình (3.13). Tất cả chúng cần thiết để khởi tạo giá trị ban đầu cho thanh ghi Y bằng 0, và thanh ghi X là hệ số tỷ lệ xích mong muốn. Nếu không đề cập đến sự khuếch đại độ lớn biên độ của véc-tơ đầu vào, thì thanh ghi X có thể được khởi tạo giá trị ban đầu là 1, và các giá trị sine, cosine có thể được đọc trực tiếp ra các thanh ghi Y và thanh ghi X tương ứng theo thứ tự.

Tuy nhiên, độ khuếch đại có nghĩa là một số tỷ lệ xích phải được biểu diễn trước khi đi vào tính toán. Sau n lần lặp của thuật toán, nội dung của các thanh ghi sẽ là:

$$\begin{aligned} X_n &= K_n X_0 \cos(Z_0) \\ Y_n &= K_n Y_0 \sin(Z_0) \quad (3.14) \\ Z_n &= 0 \end{aligned}$$

Bởi vậy, để tính toán các giá trị sine hoặc cosine của một góc θ, thì thanh ghi Z sẽ được khởi tạo với giá trị bằng θ, Y là 0 và thanh ghi X là K_nđể tính độ khuếch đại. Hình 3.2 biểu diễn quá trình CORDIC tính toán sine và cosine. Véc-tơ khởi tạo được gán khi cần. Thanh ghi X tương ứng với giá trị cosine theo tỷ lệ và thanh ghi Y tương ứng với giá trị sine theo tỷ lệ. Chúng ta có thể xác định giá trị không theo tỷ lệ bằng cách chia giá trị cuối cùng cho $K_{10} = 1.6468$. Ví dụ là sau 11 lần lặp, thuật toán đạt được:

$$\sin(30^o) = \frac{X_{10}}{K_{10}} = \frac{0.8230}{1.6468} = 0.4998$$

Phương pháp tính sine và cosine cũng giống với phép biến đổi tọa độ cực Đề-Các. Vì phép biến đổi được định nghĩa là:

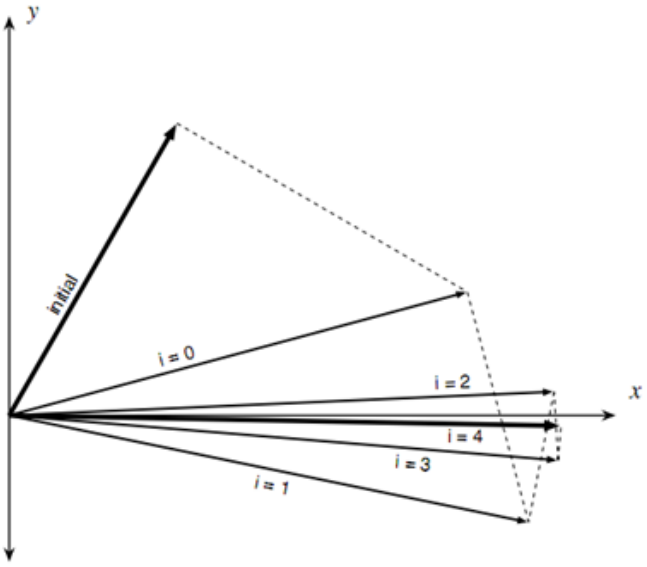
(3.15) $x = r\cos\theta$

$y = r\sin\theta$

Tất cả các hệ số cần để biểu diễn phép biến đổi là đưa θ vào Z, đưa r$K_n$vào X và đưa 0 vào Y.

3.4.2. Vectoring Mode

i	*X*	*Y*	*Z*
—	**1.0000**	**1.7321**	**0.0000°**
0	2.7321	0.7321	45.0000°
1	3.0981	–0.6340	71.5651°
2	3.2566	0.1405	57.5288°
3	3.2741	–0.2665	64.6538°
4	3.2908	–0.0619	61.0775°
5	3.2927	0.0409	59.2876°
6	3.2934	–0.0105	60.1828°
7	3.2935	0.0152	59.7351°
8	3.2935	0.0024	59.9589°
9	3.2935	–0.0041	60.0709°
10	**3.2935**	**–0.0009**	**60.0149°**



Hình 3.3: Chế độ Vectoring của thuật toán CORDIC

Trong Vectoring Mode, các phương trình sử dụng để cập nhật các thanh ghi giống nhau, trừ hàm số xác định sự thay đổi hướng quay. Trong Vectoring Mode, thuật toán thử gán véc-tơ đầu vào với trục x-, điều đó nghĩa là giá trị trong thanh ghi Y sẽ hội tụ về 0. Hàm quyết định chế độ quay là:

(3.16) $$d_i = \begin{cases} +1, nếu\ Y_i < 0 \\ -1, nếu\ Y_i \geq 0 \end{cases}$$

Hình 3.3 biểu diễn véc-tơ đầu vào hội tụ như thế nào trên trục x- qua mỗi lần lặp của thuật toán. Như trong Rotation Mode, độ lớn biên độ của véc-tơ tăng lên qua mọi lần lặp của thuật toán. Dấu của Y được dùng để xác định hướng quay, mục tiêu là đưa giá trị về 0. Trong ví dụ này, góc của véc-tơ đầu vào so với trục x- được tính và tìm thấy trong thanh ghi Z. Vì không phải là góc tỷ lệ xích, nên kết quả là góc đúng. Sử dụng hàm quyết định mới, sau n lần lặp của thuật toán, các thanh ghi sẽ là:

$$X_n = K_n\sqrt{X_0^2 + Y_0^2}$$

(3.17) $$Y_n \approx 0$$

$$Z_n = Z_0 + \tan^{-1}\left(\frac{Y_0}{X_0}\right)$$

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

Các phần dưới đây tiếp tục thảo luận về các hàm có thể được tính bởi Vectoring Mode.

3.4.2.1. Arctangent

Như đã thấy trong phương trình (3.17), hàm arctangent được tính thực chất ở thanh ghi Z trong Vectoring Mode. Để tính arctangent của một góc α, thì thanh ghi Z phải được khởi tạo giá trị ban đầu là 0, để hệ số Z_0 trong phương trình (3.17) bị khử đi và góc α phải được biểu diễn là một tỉ số của hai giá trị trong thanh ghi X và thanh ghi Y. Có thể khởi tạo giá trị ban đầu X là 1.0 và Y là α, như được thực hiện trong hình (3.3). Kết quả của arctan($\sqrt{3}$) được tính chính xác là 60^o.

(3.18) $Z_n = Z_0 + \tan^{-1}(\frac{Y_0}{X_0})$

$Z_n = 0 + \tan^{-1}(\sqrt{3})$

$Z_n = 60^o$

3.4.2.2. Biên độ véc-tơ và phép biến đổi Cực Đề-Các

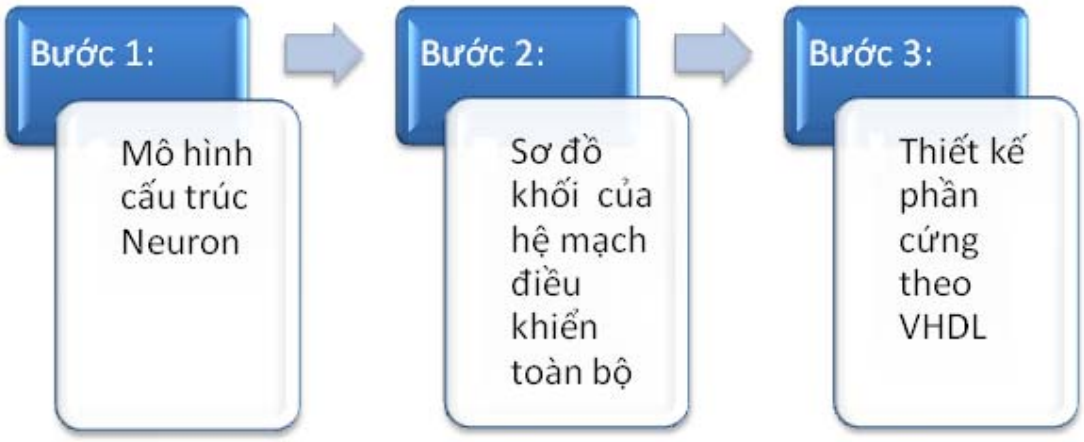
Như những phần đề cập bên trên, giá trị cuối cùng trong thanh ghi X gồm có biên độ tỷ lệ của véc-tơ đầu vào. Đặc điểm này kết hợp với phép tính thực sự của hàm arctangent tại cùng thời điểm nghĩa là kiểu vectoring CORDIC là một phép biến đổi tọa độ Đề-Các sang tọa độ cực, chỉ có kiểu Rotation là một phép biến đổi tọa độ cực sang tọa độ Đề-Các. Nhớ lại phương trình biến đổi tọa độ cực Đề-Các:

(3.19) $r = \sqrt{x^2 + y^2}$

$\theta = \tan^{-1}\frac{y}{x}$

Giá trị của r được tính trong thanh ghi X, và θ trong thanh ghi Z.

CHƯƠNG 4: THỰC HIỆN MÔ HÌNH NEURON NETWORK TRÊN FPGA



Hình 4.1: Trình tự của việc phần cứng hóa

Bước 1: Xây dựng mô hình cấu trúc Neuron

Bước 2: Xây dựng hình vẽ theo khối của hệ mạch điều khiển toàn bộ

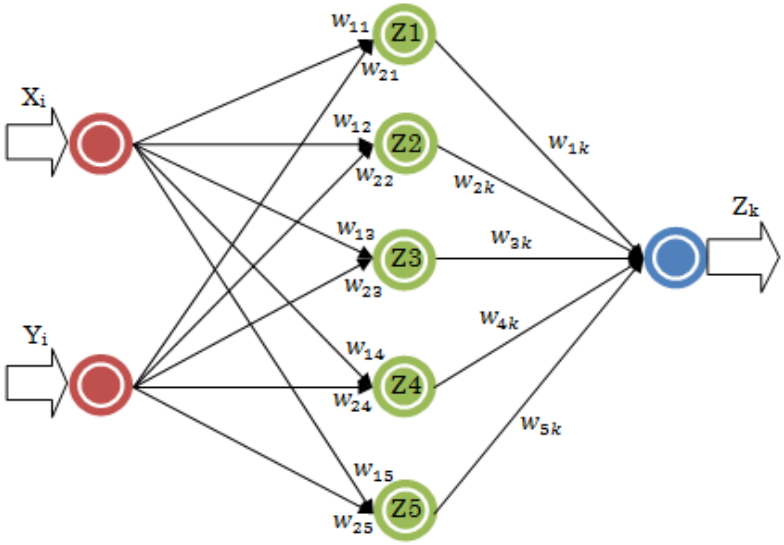
Bước 3: Thiết kế phần cứng theo VHDL

Trong chương này, học viên sẽ xây dựng mô hình NN dùng để nhận dạng một điểm ở trong hay ngoài tam giác và phần cứng sẽ được thực hiện trên FPGA. Phần tam giác này sẽ được mô phỏng bằng lập trình C cho các hệ số đã được luyện mạng và kiểm thử trước đó. Từ các hệ số này sẽ triển khai mô hình NN được phần cứng hóa gồm có hai đầu vào dữ liệu Xi và Yi là tọa độ của điểm cần xác định, năm nút lớp trung gian tính toán và một đầu ra cho biết kết quả là tọa độ Xi, Yi đó là trong hay ngoài tam giác.

Mục đích của chương này là xây dựng được các bộ cộng, bộ nhân phù hợp và phần cứng hóa hàm kích hoạt Sigmoid bằng hàm thay thế sử dụng thuật toán CORDIC để tính toán các giá trị lượng giác sin, cos để rồi cuối cùng ghép lại thành mô hình hoàn chỉnh phục vụ chức năng nhận dạng được đặt ra. Việc sử dụng mô hình NN để triển khai thực hiện trên FPGA là do khả năng linh hoạt về cấu hình và

xử lý song song vốn có của chúng. Khi thực hiện sẽ rất dễ dàng và thuận lợi, việc cấu hình nhanh, thời gian thực hiện ngắn.

4.1. Cấu trúc Neuron của Neural Networks



Hình 4.2: Cấu trúc Perceptron đơn giản

Công thức tính của xử lý trình tự có hướng như sau:

Giá trị đầu vào: Xi, Yi giữ nguyên, không làm gì và đưa luôn vào.

Lớp giữa: $Z_j = f(\sum_{j=1}^{5}(X_i w_{1j} + Y_i w_{2j}))$

(4.1)

Cụ thể:

$$Z_1 = f(X_i w_{11} + Y_i w_{21})$$

$$Z_2 = f(X_i w_{12} + Y_i w_{22})$$

$$Z_3 = f(X_i w_{13} + Y_i w_{23})$$

$$Z_4 = f(X_i w_{14} + Y_i w_{24})$$

$$Z_5 = f(X_i w_{15} + Y_i w_{25})$$

Giá trị đầu ra: $Z_k = f(\sum_{j=1}^{5} Z_j w_{jk})$

(4.2)

Cụ thể:

$$Z_k = f(Z_1 w_{1k} + Z_2 w_{2k} + Z_3 w_{3k} + Z_4 w_{4k} + Z_5 w_{5k})$$

Trong đó,

X_i, Y_i: Giá trị đầu vào của lớp đầu vào

w_{1j}, w_{2j}: weight giữa lớp đầu vào và lớp giữa

Z_j: Giá trị đầu ra của lớp đầu ra

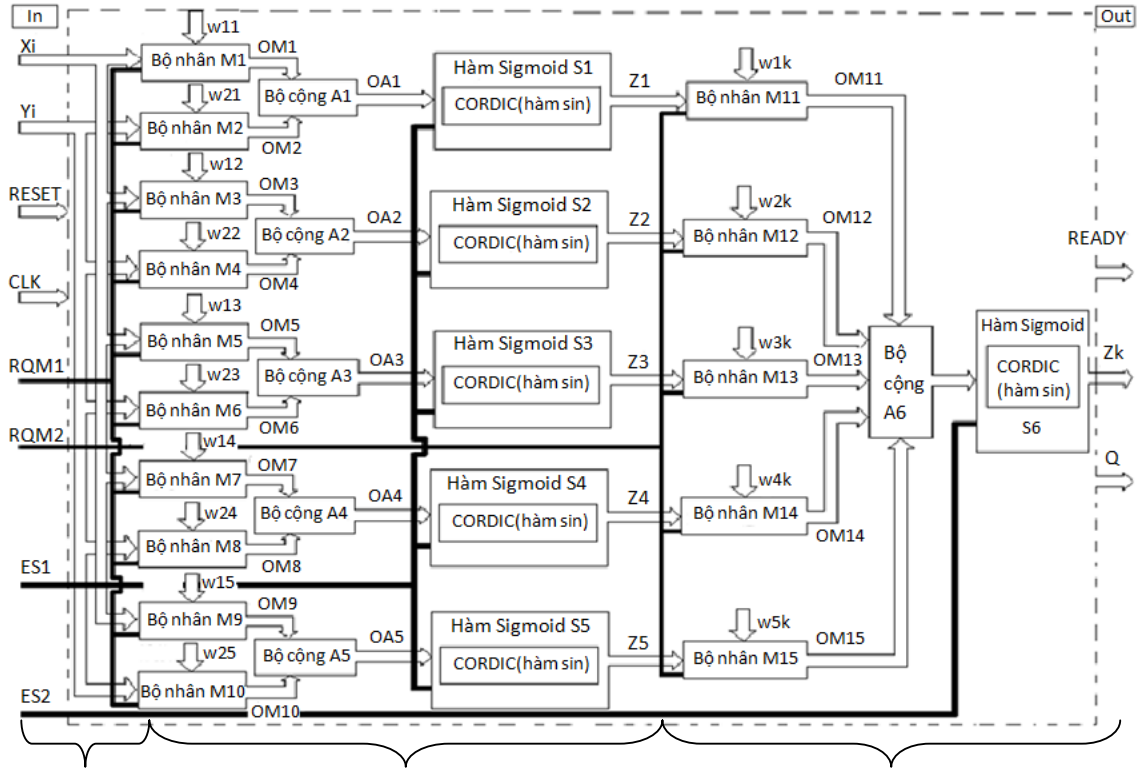
$f()$: Hàm Sigmoid (hàm kích hoạt)

w_{jk}: weight giữa lớp giữa và lớp đầu ra

Z_k:Giá trị đầu ra của lớp đầu ra

Như hình vẽ 4.2, trong việc phần cứng hóa NN cần thiết kế, chúng ta cần thiết kế bộ nhân, bộ cộng, hàm chức năng Sigmoid, CORDIC và mạch điều khiển tổng. Vì vậy, trong phương pháp nhận dạng graphic pattern, đã xây dựng hình vẽ theo khối của hệ mạch điều khiển tổng.

4.2. Hệ mạch điều khiển toàn bộ của xử lý trình tự có hướng



Hình 4.3: Sơ đồ khối phần cứng mạng nơ-ron

Mạch điều khiển tổng của xử lý trình tự có hướng của Hardware Neural Network như công thức đã trình bày ở (4.1)(4.2) được miêu tả ở hình vẽ 4.1. Trong mạch điều khiển tổng chúng ta coi tọa độ (Xi, Yi) của điểm là tín hiệu đầu vào. Thêm vào đó, trên phần cứng thực tế, để điều khiển toàn bộ mạch, chúng ta sử dụng thêm RESET, CLK, đồng thời chuẩn bị tín hiệu tính toán yêu cầu (RQM1, RQM2) hoặc tín hiệu tính toán cho phép của phần hàm Sigmoid của Multiplier.

Mặt khác, trong thời gian mạch điều khiển tổng, để xác định rõ kết quả Zk được xuất ra ở Block thứ mấy, chúng ta sử dụng tín hiệu đầu ra READY và biến điều khiển trạng thái Q.

Đây là đường đi nhận biết graphic pattern. Vì vậy, chúng ta giả sử nhập tọa độ (Xi, Yi) của điểm bất kì trên hệ trục tọa độ.

Tiếp theo, sử dụng hệ số từ tầng đầu vào đến tầng giữa wij (w11, w21, ..., w15, w25), như đầu vào của bộ nhân Mi (M1,M2,M3....M10), đầu ra của bộ nhân OMi (OM1, OM2, ..., OM10) coi như đầu vào của bộ cộng Ai (A1, A2, ..., A5). Tiếp theo, coi đầu ra của bộ cộng OAi (OA1, OA2, ..., OA10) như là một đầu vào của mạch chức năng hàm Sigmoid Si (S1, S2, …, S5). Trong khối mạch chức năng hàm Sigmoid này có khối mạch điện chức năng CORDIC. Sau đó, có thể thu được các giá trị đầu ra của khối mạch chức năng Sigmoid (Z1, Z2, Z3, Z4, Z5).

Các giá trị đầu ra ở phần trước của khối mạch hàm chức năng Sigmoid (Z1, Z2, Z3, Z4, Z5) là đầu vào của bộ nhân Mjk (M11, M12, …, M15) được thể hiện như hình vẽ sơ đồ khối ở trên.

Tiếp theo, sử dụng hệ số từ tầng giữa đến tầng đầu ra wjk (w1k, w2k, ..., w5k), như đầu vào của bộ nhân Mjk (M11, M12, ..., M15), đầu ra của bộ nhân OMjk (OM11, OM12, ..., OM15) coi như đầu vào của bộ cộng A6. Sau đó, đầu ra của bộ cộng OA6 là đầu vào của khối mạch chức năng Sigmoid S6. Trong khối mạch chức năng hàm Sigmoid này có khối mạch điện chức năng CORDIC. Tiếp theo, có thể thu được các giá trị đầu ra của khối mạch chức năng Sigmoid là Zk.

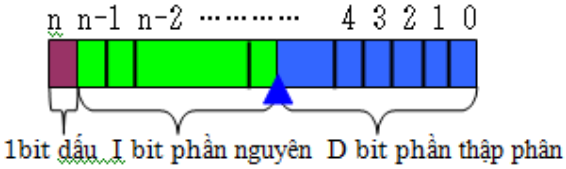
4.3. Thiết kế phần cứng với VHDL

Để phần cứng hóa đồ thị khối trên, cần có bản thiết kế mạch. Ở đây, chúng ta sử dụng VHDL để làm việc đó.

Nói cách khác, chúng ta cần thiết kế bộ nhân, bộ cộng, hàm chức năng Sigmoid, CORDIC và mạch điều khiển tổng đã được chỉ ra ở sơ đồ khối trên.

4.3.1. Bộ nhân

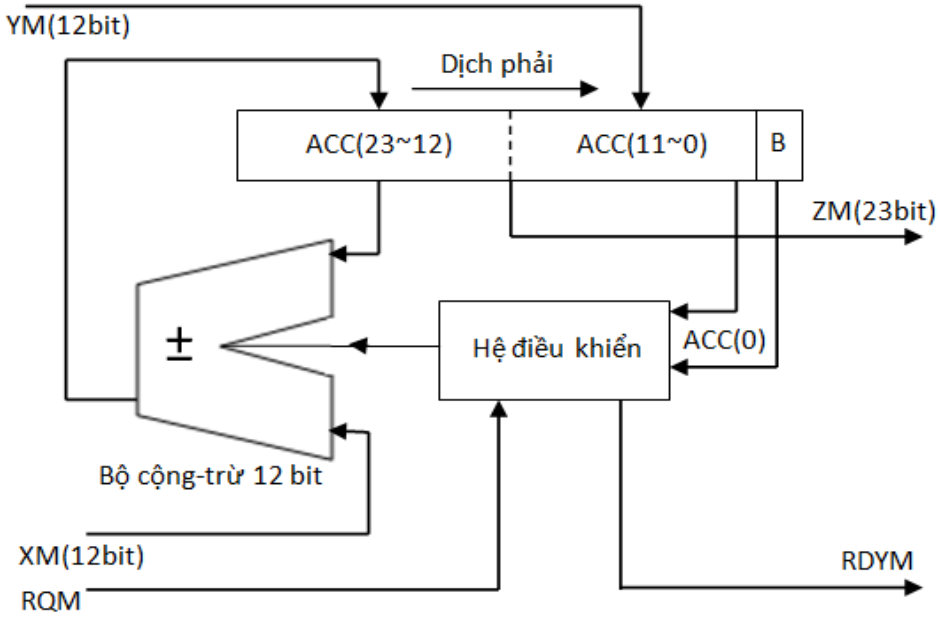
Vì các hệ số weight w1j, w2j, wjk bao gồm số âm và số thập phân, nên trong thiết kế mạch đã thiết kế bộ nhân số bù 2 dấu chấm tĩnh. Ngoài ra,về việc sử dụng cổng logic thì bộ nhân nối tiếp ít được sử dụng hơn bộ nhân song song.



Hình 4.4: Sơ đồ chiều dài của một số nhị phân

Quy ước bit dấu: 0 Số dương 1 Số âm

Bên cạnh đó, trên cơ sở kết quả nhân của hệ số và tọa độ đầu vào, cần quyết định sẽ sử dụng bao nhiêu bit trong I bit của phần số nguyên. Và trong D bit của phần thập phân, số lượng bit sử dụng thích hợp sẽ được xác định bằng giải thuật CORDIC (xem chi tiết ở phần 4.3.4).

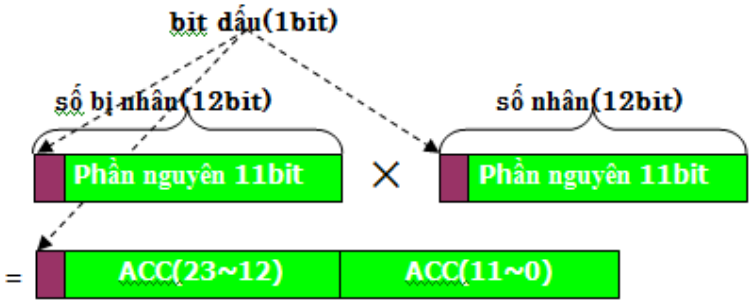


Hình 4.5: Bộ nhân song song số nguyên bù 2

ACC(ACCumulator): Biểu thị thanh ghi tổng

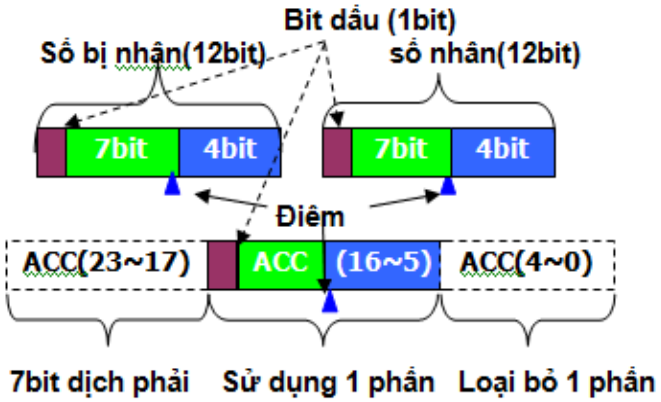
M(Multiplier): Kí hiệu bộ nhân

Quan hệ giữa số bị nhân, số nhân và tích trên bộ nhân song song số nguyên bù 2 được biểu thị như hình dưới đây:



Hình 4.6: Trường hợp nếu không có phần bit thập phân

Trường hợp nhân hai số có 7 bit phần nguyên:



Hình 4.7: Trường hợp nhân 2 số 7 bit phần nguyên

Ở đây, độ dài bit của tích số tăng lên nhiều lần nên có thể làm tròn phần bit thấp, chỉ lấy phần bít cao như là tích số. Tuy nhiên, với phần nguyên, dựa trên 7 bit và cộng trừ của XM,YM, vì có phương trình tính khác nhau nên kết quả của bộ nhân theo như dưới đây:

Trường hợp1: Trường hợp XM,YM cùng âm hay cùng dương. Phương trình tính giá trị ZM của bộ nhân:

ZM<=ACC(16 downto 5) - 1;

Trường hợp 2: Trường hợp XM,YM một âm, một dương. Phương trình tính giá trị ZM

ZM<=ACC(16 downto 5) + 1;

Kết quả thực nghiệm:

Điều khiển thời gian của đồ thị chuyển tiếp trạng thái hình 4.8 và phụ lục [A] biểu diễn trên hình 4.9.

Trong hình 4.9, Khi RESETM=1 và RQM=0, trạng thái ko thay đổi. Tiếp theo, khi RESETM=0 và RQM= 1, trạng thái Q= 0, tín hiệu yêu cầu nhân RQM= 1, bắt đầu thực hiện tính toán. Trạng thái biến đổi từ 1~11,dựa trên bít thấp nhất ACC và giá trị của thanh ghi B để điều khiển tính toán và quản lí.

Cuối cùng, trạng thái Q=12, RDYM=1, kết thúc tính, chờ đến khi trạng thái Q=13, RQM=0, rồi chuẩn bị yêu cầu thực hiện nhân tiếp theo và đưa mạch trở về trạng thái ban đầu (Q=0).

Ngoài ra, ví dụ thực tế dưới đây như là kết quả giải thích nội dung trong phần 4.1.

$(X = 2.6_{10} = 00000010.1010_{1)}) > 0$

$(X - 3.5_{10} - 00000011.1000_{1)}) > 0$

Vì vậy, kết quả có như sau:

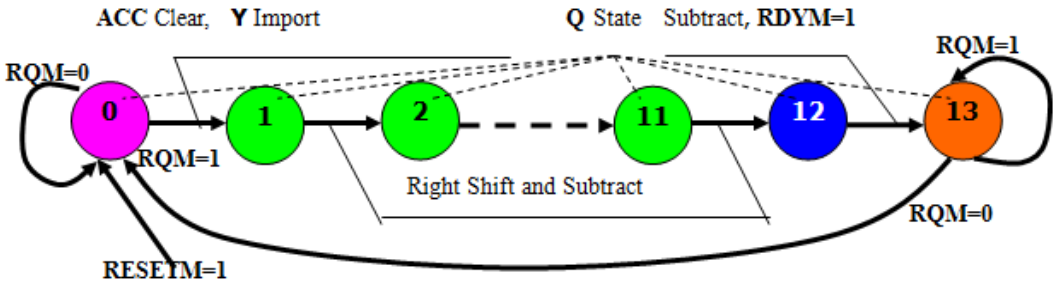
$ACC = 0000000\ \underbrace{000010010011}\ 00000$

ACC(16 downto 5)

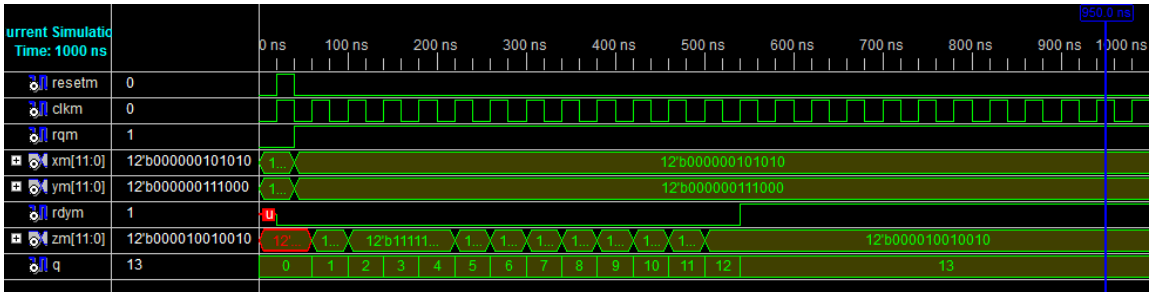
ZM<=ACC(16 downto 5) – 1 = 00001001.0011 – 1 = $00001001.0010_{\ 2} = 9.125_{10}$

Một mặt, giá trị thực là: 2.6_{10} x 3.5_{10} = 9.1_{10}

Tuy nhiên khi chuyển từ số thập phân sang số nhị phân, không sử dụng quá trình biến đổi thông thường mà sử dụng quá trình chuyển đổi hệ số đặc biệt bằng CORDIC.



Hình 4.8: Sơ đồ trạng thái



Hình 4.9: Một ví dụ về điều khiển thờigian

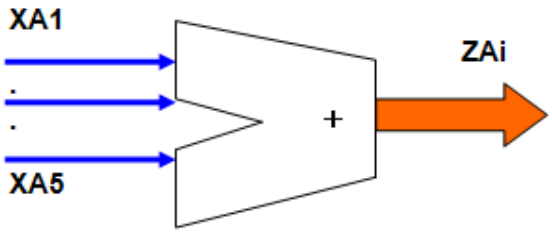
4.3.2. Bộ cộng

●Bộ cộng 2 đầu vào

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

entity Adder_12bit_Input2 is
    Port(
   XA1,XA2:in std_logic_vector(11 downto 0);
   ZAI2:out    std_logic_vector(11 downto 0)
      );

end Adder_12bit_Input2;
architecture Behavioral of dder_12bit_Input2 is
begin
   ZAI2<=XA1+XA2;
end Behavioral;
```

●Bộ cộng 5 đầu vào

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

entity Adder_12bit_input5 is
    Port(XA1,XA2,XA3,XA4,XA5:
          in std_logic_vector(11 downto 0);
          ZAI5:out std_logic_vector(11 downto 0)
      );

end Adder_12bit_input5;
architecture Behavioral of A2dder_12bit_input5 is
begin
   ZAI5<=XA1+XA2+XA3+XA4+XA5;
end Behavioral;
```

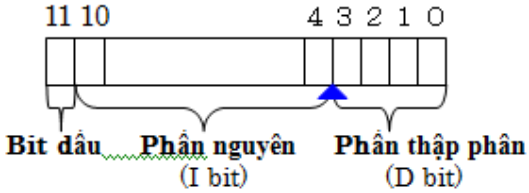


Hình 4.10: Bộ cộng 5 đầu vào số bù 2

4.3.3. Giải thuật CORDIC

Vì sử dụng số bù 2 dấu phẩy tĩnh nên cần chuyển đổi hệ thập phân sang hệ nhị phân. Tuy nhiên không sử dụng phương pháp chuyển đổi thông thường mà phải sử dụng phương pháp chuyển đổi đặc biệt dựa trên CORDIC.

4.3.3.1. Phương pháp chuyển đổi giữa hệ số thập phân và nhị phân CORDIC



Hình 4.11: Khuôn dạng dữ liệu

Theo như hình 4.11, trong 12 bits dữ liệu có 4 bit thập phân. Và như vậy, khi giá trị thực là 1.0000, vì thế nên trường hợp chuyển giá trị dữ liệu dấu phẩy thập phân sang số thập phân sẽ tuân theo công thức dưới đây:

$$(\text{Số thập phân})_{10} \times 16 = (\text{Số thập phân})_{2}$$

4.3.3.2. Mạch tính hàm sin,cos dựa trên giải thuật CORDIC

Theo như hình 4.12, khi RESETM=1 và RQM=0, đầu ra ZM=0, trạng thái không biến đổi, tiếp theo khi RESETM=0 và RQM=1, trạng thái Q=0, tín hiệu yêu cầu nhân RQM= 1, bắt đầu thực hiện tính toán. Trạng thái biến đổi từ 1~5, dựa trên bít thấp nhất ACC và giá trị của thanh ghi B để điều khiển tính toán và quản lí. Cuối cùng, trạng thái Q=6, RDYM=1, kết thúc tính, chờ đến khi trạng thái Q=7, RQM=0,

rồi chuẩn bị yêu cầu thực hiện nhân tiếp theo và đưa mạch trở về trạng thái ban đầu (Q=0).

Ví dụ thực tế dưới đây cũng giống như là kết quả giải thích cho nội dung trong phần 4.3.1

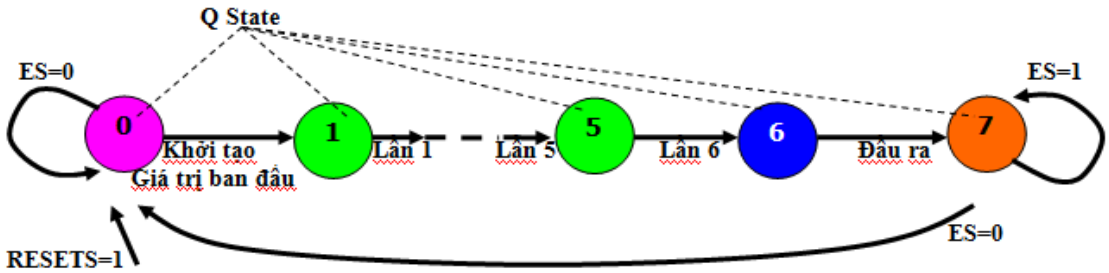
Góc nhập $TH = \frac{\pi}{2} = 1.57080_{10} = 00000001.1001_2$

Kết quả tính toán bằng CORDIC:

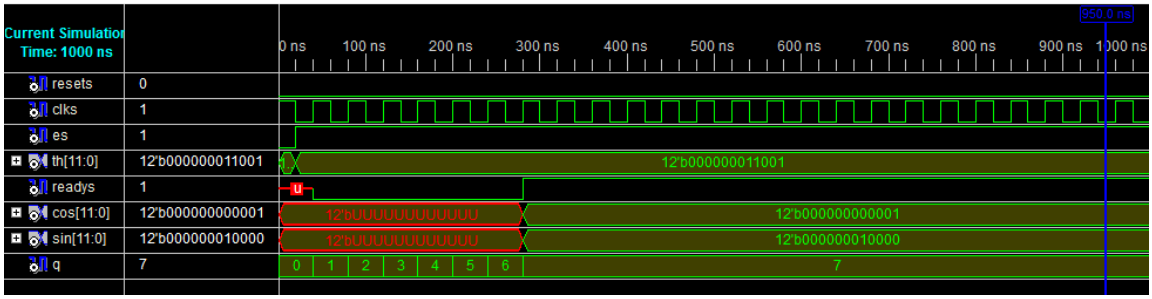
$$\cos\left(\frac{\pi}{2}\right) = 00000000.0001 = 0.0625 \approx 0$$

$$\sin\left(\frac{\pi}{2}\right) = 00000001.0000 = 1$$

Kết quả trên cho thấy độ chính xác của phép tính rất cao. Số lượng bit thập phân càng nhiều thì kết quả càng chính xác.



Hình 4.12: Sơ đồ chuyển dịch trạng thái



Hình 4.13: Một ví dụ về điều khiển thời gian

Tham khảo chương trình trong phụ lục [B]. Đoạn trong phần [--1] là phần tính toán được tiến hành để kiểm tra mối qua hệ lớn nhỏ giữa Z và Th. Do đó dữ liệu đưa vào không chỉ sử dụng số nguyên mà còn sử dụng biểu diễn số nhị phân. Bit dấu từ W là MSB (Most Significant Bit) nếu bằng 1 thì W là số âm và Z<TH.

Tiếp theo, đoạn dưới đây là đẳng thức tính toán thuyết minh bằng lí luận thuật toán CORDIC

Đoạn này chỉ ra rằng để kiểm tra mối quan hệ lớn bé của Z và Th thì phải tiến hành tính toán. Đó là: việc sử dụng các dữ liệu nhập vào không chỉ là các số Integer mà còn áp dụng cả việc hiển thị những số nhị phân. Tiếp theo, đoạn sau đây, theo như logic của giải thuật CORDIC giải thích cách tính toán:

$$x_i - x_{i-1} - s_{i-1}(\frac{1}{2^{i-1}})y_{i-1}$$

$$y_i = y_{i-1} - s_{i-1}(\frac{1}{2^{i-1}})x_{i-1}$$

$$s_{i-1} = \begin{cases} +1 & (\theta_{i-1} \leq \theta) \\ -1 & (\theta_{i-1} > \theta) \end{cases}$$

Ở đây nhìn vào [--2] và [--3], X và Y chỉ cần tính toán dịch sang bên phải (Q-1) bit. Lần lượt XX và YY tạo thành một mạch. Bởi vì ở đây có sự chuyển đổi của từng bit vector dữ liệu tín hiệu nhập vào X,Y thành từng std_logic_vector.

[--4] là phần tiến trình, đó là cơ cấu kiểm soát của mạch tính toán, đó là mạch để cưỡng chế reset. Danh sách chi tiết của phần tiến trình sẽ cưỡng chế tín hiệu reset và tín hiệu clock.

[--5] trong trường hợp trạng thái 0, xảy ra

[--6] trạng thái từ 1~5…

[--7] chỉ ra bởi vì W(7) là bit nhãn, khi W(7)=1 thì Z<TH.

[--8] trạng thái 6…. Cờ READYS sẽ dựng lên, …

[--9] trạng thái 7…. ở đây thì nếu ES=0 thì cờ READYS sẽ hạ xuống () và trả lại trạng thái về trạng thái ban đầu.

4.3.4. Phần hàm số Sigmoid

Ở phần trước, đã giải thích về hàm số Sigmoid. Nhưng ở đây, khi mà thiết kế mạch thực tế, ta sẽ giải thích về cách sử dụng nó như thế nào. Xem hình 4.14 và hình 4.15, [-π/2, π/2] theo như CORDIC: Biểu đồ hàm sin là biểu đồ hàm sin bình thường và đồng dạng nhưng trong trường hợp (-∞,-π/2) và (π/2, ∞) thì khác nhau. Thực tế thì hàm sin bình thường và đồng dạng dùng VHDL để dự định thiết kế

mạch (theo như CORDIC về hàm sin) nhưng nếu xem hình 4.16 trong trường hợp (-∞,-π/2) và (π/2, ∞) sẽ hiểu được sự đồng dạng về tính tự động trong hàm Sigmoid. Hơn thế nữa, nói về hàm thay thế hàm Sigmoid, có trong rất nhiều hàm, theo như giải thuật CORDIC về hàm sin đã tìm thấy điều thích hợp nhất.

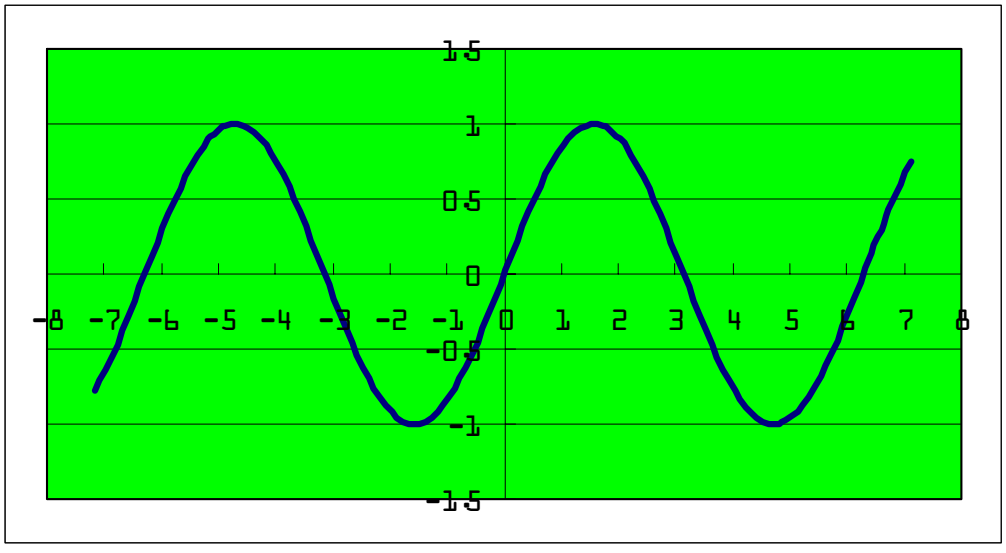
Về phía hàm Sigmoid và hàm thay thế cho hàm Sigmoid, biểu thị cho các hình thức dưới đây.

Hàm Sigmoid : $f_x = \frac{1}{1+e^{-x}}$

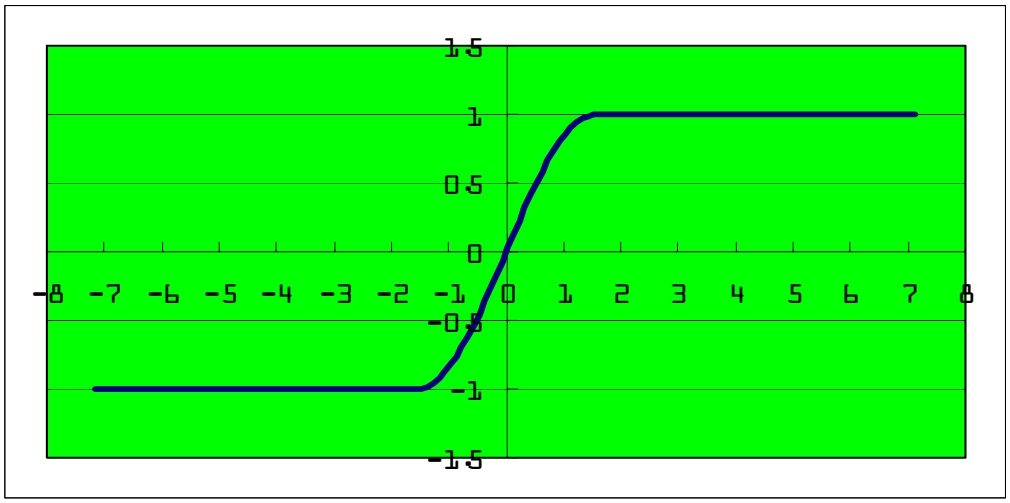
Hàm thay thế hàm Sigmoid:

$$f() = \begin{cases} 0 & (x \le -\frac{\pi}{2}) \\ \frac{1+\sin x}{2} & (-\frac{\pi}{2} \le x \le \frac{\pi}{2}) \\ 1 & (\frac{\pi}{2} \le x) \end{cases}$$

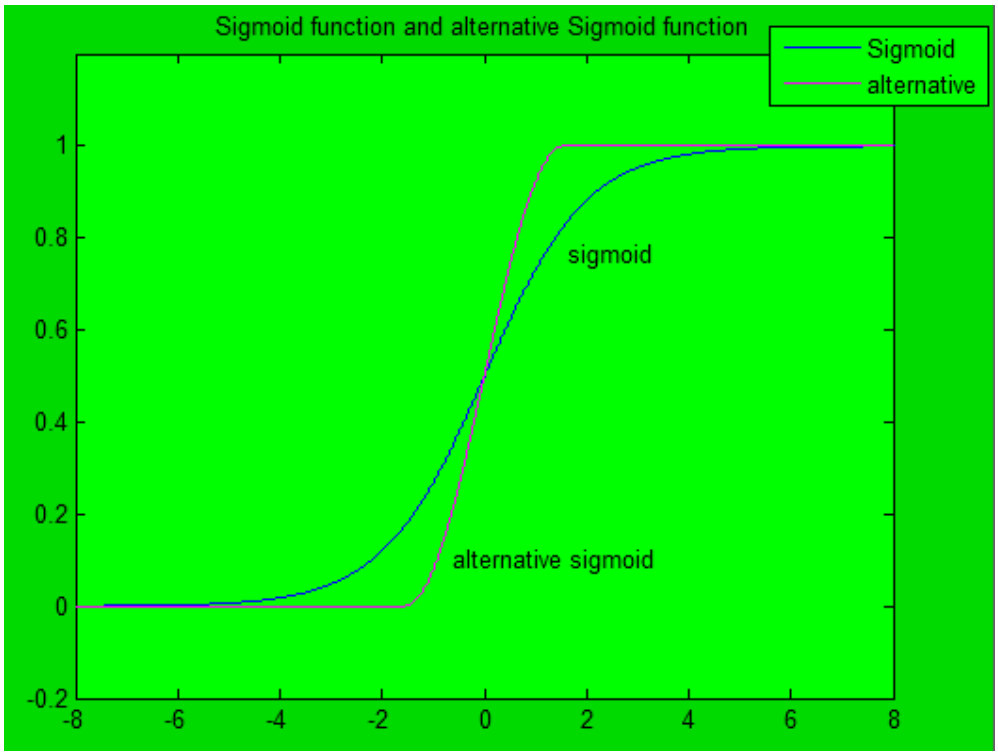
Tuy nhiên, sinx không phải là hàm sinx thông thường, mà là "hàm sin xây dựng theo CORDIC"



Hình 4.14: Hàm sin thông thường



Hình 4.15: Hàm sin xây dựng theo CORDIC



Hình 4.16: Đồ thị hàm Sigmoid

Theo hình 4.15, do trên hình hàm sin xây dựng theo CORDIC

$$\sin x = \begin{cases} -1 & (x \le -\frac{\pi}{2}) \\ 1 & (\frac{\pi}{2} \le x) \end{cases}$$

Giá trị thay thế của hàm Sigmoid sẽ tự động trở thành:

$$f() = \frac{1+\sin x}{2} = \begin{cases} 0 & (x \le -\frac{\pi}{2}) \\ 1 & (\frac{\pi}{2} \le x) \end{cases}$$

Vì thế, không cần thiết kế disctinction circuit, chỉ cần thiết kế một trường hợp $\left(-\frac{\pi}{2} < x < \frac{\pi}{2}\right)$. Cách thiết kế circuit đó như sau:

```
**************Code 4.3.4***************
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Sigmoid_Cordic_Sin_Function is
   Port ( RESETS,CLKS,ES: in std_logic;
          READYS: out std_logic;
          WA: in std_logic_vector(11 downto 0);
          OS: out std_logic_vector(11 downto 0)
        );
end Sigmoid_Cordic_Sin_Function;

architecture  Behavioral of Sigmoid_Cordic_Sin_Function is
   component Cordic_Sin_Function_12bit
      Port ( RESETS,CLKS,ES: in std_logic;
             READYS: out std_logic;
             TH: in std_logic_vector(11 downto 0);
             COS,SIN: out std_logic_vector(11 downto 0)
            );
   end component;

signal SIN:std_logic_vector(11 downto 0);

begin
   CORDIC:Cordic_Sin_Function_12bit
port map(RESETS,CLKS,ES,READYS,WA,open,SIN);
   OS<=to_stdlogicvector(to_bitvector (16+SIN) sra 1);    --"00000001.0000"=16(Integer)

end Behavioral;
```

Trong phần Entity, về tín hiệu vào ra của Port, RESETS là tín hiệu reset, CKLS là tín hiệu Clock, ES là tín hiệu cho phép khởi đầu tính toán, READYS là cờ báo hiệu kết thúc tính toán.

Trong phần Architecture, tín hiệu vào ra của component hoàn toàn giống với tín hiệu của cú pháp VHDL của Cordic_sin_Function. Đó là vì CORDIC có thể sử dụng như một thành phần trong phần Sigmoid.

Trong phần signal, SIN là biến số của hàm tính toán Sigmoid. Đó là vì, kết quả tính toán hàm sin nhờ CORDIC là tín hiệu đầu ra, nên không thể sử dụng làm biến số cho phần portmap.

Không cần nhân trực tiếp $f() = \frac{1+\sin x}{2}$, ta có thể biểu diễn như sau vì 1 = "00000001.0000" = 16 nên cần thiết phải biến đổi bit_vector sang std_logic_vector. Thêm nữa, để thực hiện (1+sinx)/2, chỉ cần dịch bit sra (shift right arithmetic) sang phải một bit là được. Sau đây sẽ chỉ ra một ví dụ:

Giả sử 8.0 = "00001000.0000", muốn tính 4 = $\frac{8}{2^1}$, chỉ cần dịch 1 bit sang phải, ta có "00000100.0000" = 4.

4.3.5. Toàn bộ mạch điều khiển

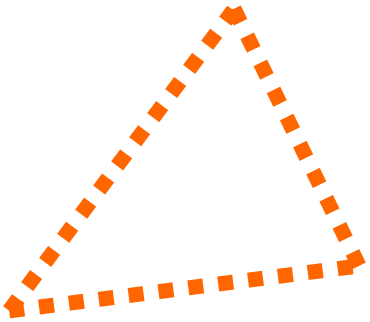
Như đã chỉ ra trên toàn bộ mạch điều khiển vẽ trên hình 4.3, có 15 bộ phận tính toán (M1~M16), máy cộng và mạch của hàm Sigmod (A1~A6), (S1~S6). Trong đó, bộ nhân và hàm Sigmoid thì tất cả các mạch giống nhau nên mỗi cái chỉ cần tạo ra một mạch. Như có thể thấy trên hình, bộ cộng của tầng trung gian và bộ cộng của tầng đầu là khác nhau, cho nên ta thiết kế hai mạch khác nhau .

Vì thế, rất dễ để thiết kế khi sử dụng mỗi mạch bộ phận như một thành phần của toàn bộ mạch điều khiển. Tham khảo phụ lục [C]

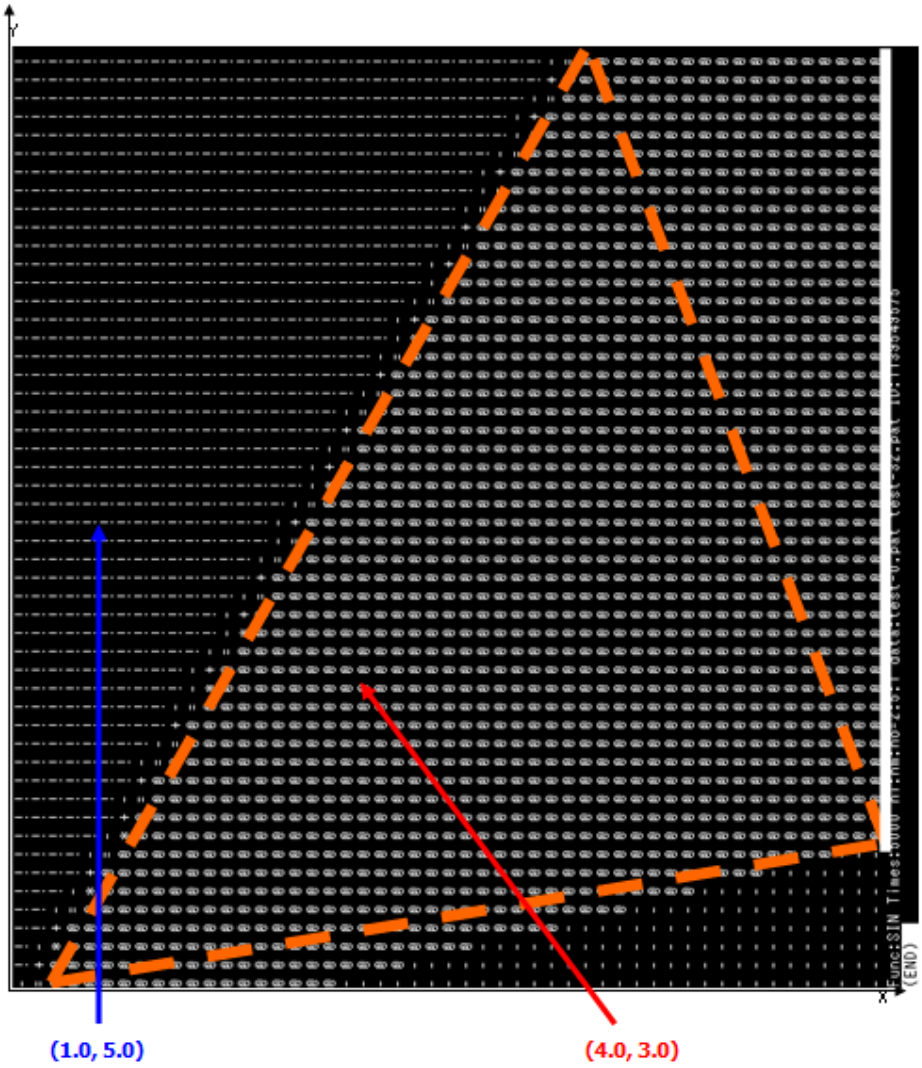
4.4. Kết quả thực hiện

Dựa vào kết quả mô phỏng của Digital Neural Network, nhập tín hiệu đầu vào phần cứng bằng điểm bất kì bên trong và bên ngoài tam giác. Đầu ra phần cứng là 1 thì xác định đó là điểm nằm bên trong tam giác, nếu đầu ra là 0 thì xác định chính xác đó là điểm ngằm ngoài tam giác.

Trên thực tế, mô phỏng của các nghiên cứu hi vọng thu được tam giác như hình 4.17, tuy nhiên kết quả thu được chỉ như hình 4.18, vì vậy khi thực nghiệm trên phần cứng, chỉ dựa vào hình vẽ đó và xác định điểm bên trong và bên ngoài của hình.



Hình 4.17: Hình dáng mong muốn

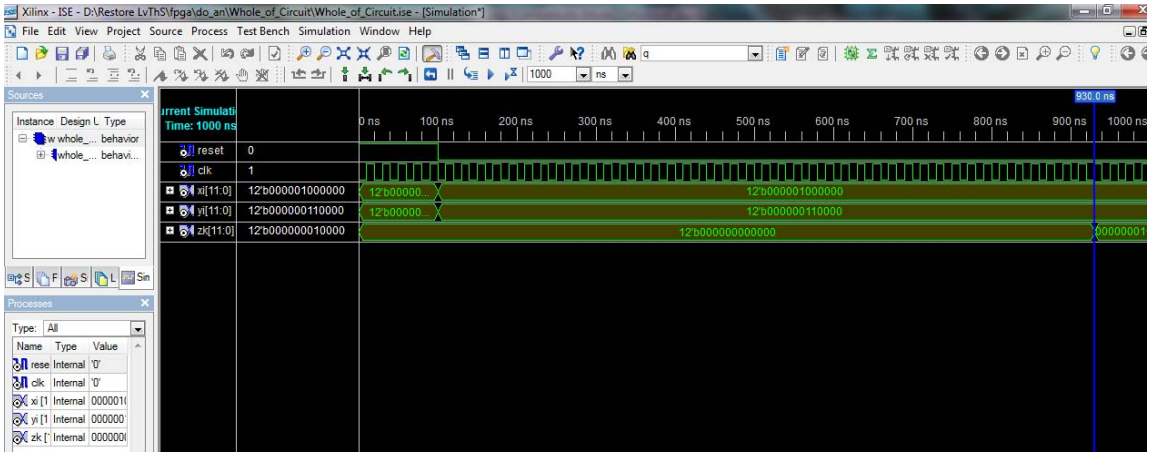


Hình 4.18: Kết quả mô phỏng hình dạng của NN số hóa

Với mỗi kí tự đại diện cho một dải giá trị tính toán đầu ra trong mô phỏng hình dáng tam giác mong muốn

Bảng 4.1: Dải giá trị của các kí tự

!	-	+	=	*	@
0	[0, 0.25]	[0.25, 0.5]	[0.5, 0.7]	[0.7, 0.85]	[0.85, 1]



Hình 4.19: Kết quả nhận dạng điểm (4, 3) ở trong tam giác

Cụ thể, như ví dụ hình 4.19, việc nhân diện điểm (4.0,3.0) nằm phía bên trong tam giác được tiến hành như sau:

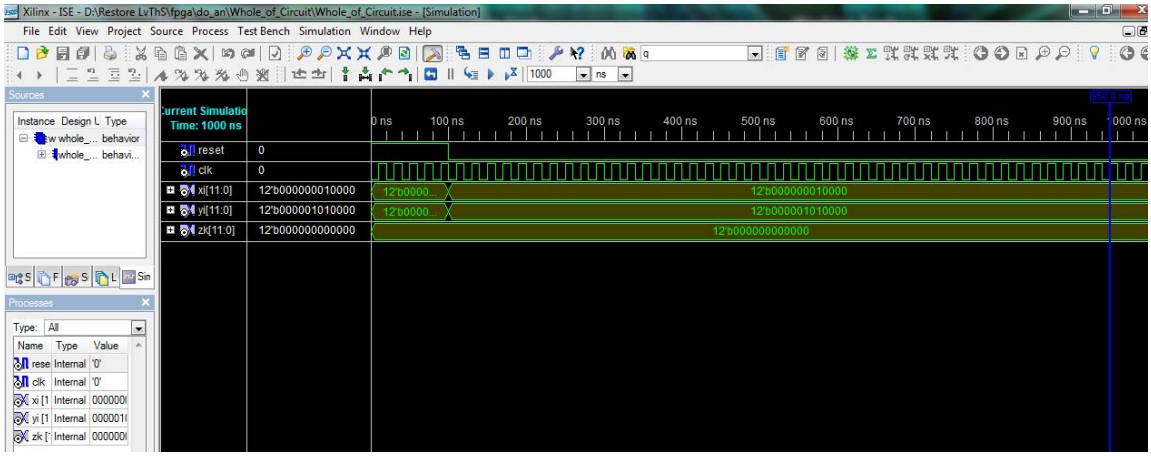
Đầu vào của phần cứng Digital Neural Network (như hình 4.19), lần này chúng ta sử dụng dữ liệu có độ dài từ 12bit (1:7:4 theo thứ tự là bit dấu: phần số nguyên: phần thập phân). Tiếp theo, sử dụng phương pháp biến đổi theo CORDIC, đổi sang số nhị phân, thiết lập cho giá trị đầu vào bằng nút có trên thiết bị thực nghiệm phần cứng, cụ thể là :

4.0=00000100.0000

3.0=00000011.0000

sử dụng như các trạng thái của switch (ON, OFF).

Dựa vào giá trị nhập vào của mô phỏng tính hệ số của Neuron, tiến hành thực nghiệm dựa vào hoạt động của hệ mạch điều khiển toàn bộ, kết quả đầu ra của hệ mạch là **Zk=00000001.0000=1** chúng ta xác định được đây là điểm nằm trong tam giác mô phỏng. Kết quả như hình 4.19.



Hình 4.20: Kết quả nhận dạng điểm (1, 5) ở ngoài tam giác

Ngược lại, ở hình 4.20, việc xác định điểm nằm phía ngoài (1.0, 5.0) được tiến hành như sau:

Cụ thể: **1.0=00000001.0000**

5.0=00000101.0000

Cụ thể, tính toán hệ số của Neuron kết quả trên, tiến hành thực nghiệm dựa vào hoạt động của hệ mạch điều khiển toàn bộ, ta thu được kết quả đầu ra là **Zk=00000000.0000=0**, xác định đây là điểm nằm ngoài tam giác mô phỏng. Kết quả như hình 4.20

4.5. Kết quả tổng hợp trên FPGA của Xilinx

Triển khai thiết kế NN với FPGA Spartan3E-500 của Xilinx dùng công cụ ISE cho kết quả sử dụng tài nguyên trên FPGA như bảng dưới đây:

Bảng 4.2: Kết quả sử dụng tài nguyên của NN trên FPGA

	Slices	**Flip-Flops**	**LUTs**	**IOBs**
Sử dụng	1439	712	2771	7
Sẵn có	4656	9312	9312	66
%	30	7	29	10

Như vậy, trong chương này đã trình bày về phần thực hiện và kết quả thực hiện mô hình NN trên FPGA. Qua đó thấy được việc lựa chọn mô hình NN để tiến

hành triển khai trên phần cứng là hoàn toàn hợp lý về tài nguyên sử dụng, đồng thời việc lựa chọn loại chip này của Xilinx là hoàn toàn phù hợp với thiết kế NN đã đề ra.

KẾT LUẬN

Trong bối cảnh hiện nay, khi mà các nghiên cứu đang chuyển hướng sang xây dựng các hệ thống thông minh, mạng Neuron nổi lên như một giải pháp đầy hứa hẹn. Nó thể hiện những ưu điểm nổi bật của mình so với các hệ thống khác ở khả năng mềm dẻo, linh hoạt và tính toán thô. Đây cũng chúnh là những khác biệt giữa bộ óc người với các máy thông minh nhân tạo. Nhưng cũng chính vì thế mà nó đòi hỏi một độ phức tạp rất cao trong thiết kế và cài đặt các hệ thống ứng dụng để có thể đạt được một tính năng tốt. Điểm mấu chốt của quy mô hệ thống là số lượng các Neuron và số lượng các lớp ẩn. Khả năng này sẽ được cải thiện không ngừng trong tương lai cùng với sự phát triển của các mạch tích hợp phần cứng cỡ lớn và các bộ nhớ ngày càng lớn hơn cho các phần mềm. Chính vì điều này mà mạng Neuron được coi là kỹ thuật của thế kỷ 21.

Tuy nhiên, thông qua các nghiên cứu phần mềm trước đây, có thể thấy được vấn đề quan trọng cần phải tiếp tục được nghiên cứu và giải quyết đó là vấn đề cải thiện bản thân thuật toán và kỹ thuật xử lý song song có thể đem lại những tính năng tốt hơn. Cụ thể luận văn đã sử dụng và phát triển tốt những ưu điểm xử lý song song của FPGA để triển khai một mạng Neuron nhân tạo. Sai số xuất hiện ở đây do việc xấp xỉ hàm Sigmoid và sai số lượng tử do giới hạn bit.

Sau quá trình thực hiện luận văn, học viên đã nghiên cứu và nắm được những kiến thức về mạng Neuron, cũng như việc triển khai thực tế của NN trên FPGA. Học viên đồng thời đánh giá và lựa chọn thuật toán và mô hình NN phù hợp để triển khai trên FPGA. Như vậy mục tiêu đề ra khi lựa chọn đề tài luận văn đã được học viên thực hiện đầy đủ. Tuy nhiên, những nghiên cứu về NN cũng như việc triển khai NN dựa vào phần cứng thực tế còn nhiều điểm cần tiếp tục tìm hiểu và phân tích thêm trong quá trình nghiên cứu sau này của học viên như triển khai các ứng dụng trên NN. Bản thân luận văn được xây dựng dựa trên nền tảng phần cứng và ứng dụng nó cho vấn đề nhận dạng, nó hứa hẹn sẽ có rất nhiều điều có thể tiếp tục phát triển.

TÀI LIỆU THAM KHẢO

[1] Yihua Liao (1999), "Neural Networks in Hardware - A Survey", Department of Computer Science, University of California, USA.

[2] Jihan Zhu, Peter Sutton (2003), "FPGA Implementations of Neural networks - a Survey of a Decade of Progress", Proceeding of the 13th International Conference on Field Programmable Logic and Applications.

[3] Amos R. Omondi, Jagath C. Rajapakse (2006), *FPGA Implementations of Neural Networks*, Springer, pp. 3-97.

[4] Charu Gupta (2006), "Implementation of Back Propagation Algorithm (of Neural Networks) in VHDL", Department of Electronics and Communication Engineering, Thapar Institute of Engineering and Technology, India.

[5] Douglas L.Perry (2002), *VHDL Programming by Example*, McGraw-Hill Companies, U.S.

[6] Pong P.Chu (2006), *RTL Hardware Design Using VHDL*, A John Wiley & Sons, U.S.

[7] Pong P.Chu (2008), *FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version*, Wiley-Interscience, U.S.

[8] Tim McLenegan (2006), "The CORDIC Algorithm: An Area-Efficient Technique for FPGA-Based Artificial Neural Networks", California Polytechnic State University, U.S.

[9] Aissa KHELDOUN, Djalal Eddine KHODJA, Larbi REOUFI (2010), "Sigmoid Function Approximation for ANN Implementation in FPGA Devices", WSEAS, U.S.

[10] M.T.Tommiska (2003), "Efficient digital implementation of the Sigmoid function for reprogrammable logic", Computers and Digital Techniques, pp. 403-411.

[11] Volnei A. Pedroni (2004), *Circuit Design with VHDL*, Massachusetts Institute of Technology, U.S.

[12] Howard B. Demuth, Mark Beale, Martin T. Hagan (2002), *Neural Network Design*, University of Colorado Bookstore, U.S.

[13] Karen Parnell, Nick Mehta (2002), *Programmable Logic Design Quick Start Handbook*, Xilinx, U.S.

[14] Peter Wilson (2007), *Design recipes for FPGA*, Elsevier, UK.

[15] Simon Haykin (1998), *Neural Network A Comprehensive Foundation*, Pearson Prentice Hall, U.S.

[16] Daniel Graupe (2007), *Principles of Artificial Neural Networks*, World Scientific Publishing, U.S.

[17] Bernard Widrow, Michael A. Lehr (2002), "30 years of Adaptive Neural Networks - Perceptron, Madaline, and Backpropagation", Proceeding of the IEEE.

[18] Haitham Kareem Ali, Esraa Zeki Mohammed (2010), "Design Artificial Neural Network Using FPGA", International Journal of Computer Science and Network Security, Vol.10 No.8.

[19] M. A. Bañuelos-Saucedo, J. Castillo-Hernández, S. Quintana-Thierry, R. Darnilán-Zamacona, J.Valerlano-Assem, R. E. Cervantes, R. Fuentes-González, G. Calva-Olmos, J. L. Pérez-Silva (2003), "Implementation of a Neuron Model using FPGAs", Journal Applied Research and Technology.

[20] A. Muthuramalingam, S. Himavathi, E. Srinivasan (2008), "Neural Network Implementation Using FPGA Issues and Application", International Journal of Information Technology.

[21] Rafid Ahmed Khalil (2007), "Hardware Implementation of Backpropagation Neural Networks on FPGA", University of Mosul, Iraq.

[22]Jack E. Volder (1959), "The CORDIC Trigonometric Computing Technique", The Institute of Electrical and Electronics Engineers

[23] Hoàng Mạnh Hà, Trần Thanh Phương (2007), "Hiện Thực Mô Hình Mạng Neuron Nhân Tạo Trên FPGA", Tạp chí Khoa Học và Ứng Dụng, số 4, trang 45.

PHỤ LỤC

A. Bộ nhân dấu chấm tĩnh bù 2

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

entity Multiplier_12bit is
  Port (RESETM,CLKM,RQM:in std_logic;
  RDYM:out std_logic;
  XM,YM:in std_logic_vector(11 downto 0);
  ZM:out std_logic_vector(11 downto 0)
  );
end Multiplier_12bit;

architecture Behavioral of Multiplier_12bit is
signal Q:integer range 0 to 13;
signal ACC:std_logic_vector(23 downto 0);
signal B:std_logic;
signal WA,WB:std_logic_vector(11 downto 0);

begin
ZM<=ACC(16 downto 5)+k;
WA<=ACC(23 downto 12)-XM;
WB<=ACC(23 downto 12)+XM;
  process(CLKM,RESETM) begin
     if RESETM='1' then RDYM<='0';Q<=0;
     elsif CLKM'event and CLKM='1' then
       case Q is
         when 0 =>if RQM='1'
            then Q<=1; ACC<=(23 downto 12=>'0') & YM;B<='0';end if;
         when 1 to 11 =>
               if ACC(0)='1' and B='0'
            then ACC<=WA(11) & WA & ACC(11 downto 1);
                  B<=ACC(0);
              elsif ACC(0)='0' and B='1'
            then ACC<=WB(11)& WB & ACC(11 downto 1);
                  B<=ACC(0);
              else ACC<=ACC(23)&ACC(23 downto 1);end if;
               Q<=Q+1;
         when 12 =>
               if ACC(0)='1'and B='0'
                  then ACC(23 downto 12)<=WA;
               elsif ACC(0)='0'and B='1'
                  then ACC(23 downto 12)<=WB;
               end if;
               Q<=Q+1;RDYM<='1';
         when 13 =>
```

```
                if RQM='0' then RDYM<='0';Q<=0;end if;
            when others =>null;
           end case;
          end if;
        end process;
      end Behavioral;
```

B. Chuyển đổi CORDIC thập phân sang nhị phân và ngược lại

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

entity Cordic_Sin_Function_12bit is
    Port ( RESETS : in  STD_LOGIC;
           CLKS : in  STD_LOGIC;
           ES : in  STD_LOGIC;
           READYS : out  STD_LOGIC;
           TH : in  STD_LOGIC_VECTOR (11 downto 0);
           COS : out  STD_LOGIC_VECTOR (11 downto 0);
           SIN : out  STD_LOGIC_VECTOR (11 downto 0));
end Cordic_Sin_Function_12bit;

architecture Behavioral of Cordic_Sin_Function_12bit is
   subtype CONST_type is std_logic_vector(11 downto 0);
           type ARRAY_DATA is array(0 to 4) of CONST_type;
           constant CONST: ARRAY_DATA:=ARRAY_data'(
                                    "000000001101","000000000111",--00D,007
                                    "000000000100","000000000010",--004,002
                                    "000000000001" );              --001
signal Q: integer range 0 to 7;
signal X,XX,Y,YY,Z,W: std_logic_vector(11 downto 0);
begin
W<=Z-TH;                        --1
XX<=to_stdlogicvector(to_bitvector(x) sra Q-1);--2
YY<=to_stdlogicvector(to_bitvector(y) sra Q-1);--3
  process(RESETS,CLKS) begin
    if       RESETS='1'       then       Q<=0;       READYS<='0';       SIN<="000000000000";
COS<="000000000000";       --4
                    elsif CLKS'event and CLKS='1' then
                                    if Q=0 then                    --5
                                    if ES='1' then
                                    READYS<='0';
                                                    Z<=(others=>'0');
                                                    X<="000000001010";--00A
                                                    Y<=(others=>'0');
                                                    Q<=1;
                                            end if;
                                    elsif Q<6 then                 --6
```

```
                    if W(7)='1' then                --7
                        X<=X-YY;Y<=Y+XX;Z<=Z+CONST(Q-1);
                            else
                        X<=X+YY;Y<=Y-XX;Z<=Z-CONST(Q-1);
                            end if;
                            Q<=Q+1;
                        elsif Q=6 then READYS<='1';COS<=X;SIN<=Y;Q<=7;  --8
                        else                                    --9
                    if ES='0' then Q<=0;READYS<='0';end if;
                        end if;
                end if;
  end process;
end Behavioral;
```

C. Toàn bộ mạch điều khiển

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

entity Whole_of_Circuit is
    Port (
                        RESET,CLK:in std_logic;
                        Xi,Yi:in std_logic_vector(11 downto 0);
                        Zk:out std_logic_vector(11 downto 0)
                        );
end Whole_of_Circuit;

architecture Behavioral of Whole_of_Circuit is
    component Multiplier_12bit
      Port (
                            RESETM,CLKM,RQM:in std_logic;
                            RDYM:out std_logic;
                            XM,YM:in std_logic_vector(11 downto 0);
                            ZM:out std_logic_vector(11 downto 0)
                            );
                end component;

                component Adder_12bit_Input2
                        Port (
                            XA1,XA2: in std_logic_vector(11 downto 0);
                            ZAI2:out std_logic_vector(11 downto 0)
                            );
                end component;

                component Sigmoid_Cordic_Sin_Function
                        Port (
                            RESETS,CLKS,ES: in std_logic;
                            READYS: out std_logic;
```

```
                                        WA: in std_logic_vector(11 downto 0);
                                        OS: out std_logic_vector(11 downto 0)
                                        );
                end component;

                component Adder_12bit_input5
    Port (
                                        XA1,XA2,XA3,XA4,XA5:    in    std_logic_vector(11
downto 0);
                                        ZAI5:out std_logic_vector(11 downto 0)
                                        );
  end component;

signal Q:integer range 0 to 42;
signal RQM1,RQM2,ES1,ES2,READY:std_logic;
signal
OM1,OM2,OM3,OM4,OM5,OM6,OM7,OM8,OM9,OM10,OM11,OM12,OM13,OM14,OM15
,OA1,OA2,OA3,OA4,OA5,OA6,Z1,Z2,Z3,Z4,Z5,w11,w21,w12,w22,w13,w23,w14,w24,w15
,w25,w1k,w2k,w3k,w4k,w5k:std_logic_vector(11 downto 0);

begin
--Multiplier Mi(i=1~10)

w11<="000000001001";--w11=0.5401840
M1:Multiplier_12bit port map(RESET,CLK,RQM1,open,Xi,w11,OM1);
--OM1:output of M1

w21<="000000000101";--w21=0.3211372
M2:Multiplier_12bit port map(RESET,CLK,RQM1,open,Yi,w21,OM2);
--OM2:output of M2

w12<="000000000011";--w12=0.1977752
M3:Multiplier_12bit port map(RESET,CLK,RQM1,open,Xi,w12,OM3);
--OM3:output of M3

w22<="000000001001";--w22=0.5743167
M4:Multiplier_12bit port map(RESET,CLK,RQM1,open,Yi,w22,OM4);
--OM4:output of M4

w13<="000000111110";--w13=3.9021218
M5:Multiplier_12bit port map(RESET,CLK,RQM1,open,Xi,w13,OM5);
--OM5:output of M5

w23<="100000011110";--w23=-1.8753718
M6:Multiplier_12bit port map(RESET,CLK,RQM1,open,Yi,w23,OM6);
--OM6:output of M6

w14<="100000000111";--w14=-0.4268485
M7:Multiplier_12bit port map(RESET,CLK,RQM1,open,Xi,w14,OM7);
--OM7:output of M7
```

```
w24<="000000011101";--w24=1.7927740
M8:Multiplier_12bit port map(RESET,CLK,RQM1,open,Yi,w24,OM8);
--OM8:output of M8

w15<="100001111001";--w15=-7.5608993
M9:Multiplier_12bit port map(RESET,CLK,RQM1,open,Xi,w15,OM9);
--OM9:output of M9

w25<="000001001100";--w25=4.77022408
M10:Multiplier_12bit
port map(RESET,CLK,RQM1,open,Yi,w25,OM10);--OM10:output of M10


--Adder Ai(i=1~5)
A1:Adder_12bit_Input2 port map(OM1,OM2,OA1);--OA1:output of A1

A2:Adder_12bit_Input2 port map(OM3,OM4,OA2);--OA2:output of A2

A3:Adder_12bit_Input2 port map(OM5,OM6,OA3);--OA3:output of A3

A4:Adder_12bit_Input2 port map(OM7,OM8,OA4);--OA4:output of A4

A5:Adder_12bit_Input2 port map(OM9,OM10,OA5);--OA5:output of A5


--Sigmoid function Si(i=1~5)

S1:Sigmoid_Cordic_Sin_Function
port map(RESET,CLK,ES1,open,OA1,Z1);--Z1:output of S1

S2:Sigmoid_Cordic_Sin_Function
port map(RESET,CLK,ES1,open,OA2,Z2);--Z2:output of S2

S3:Sigmoid_Cordic_Sin_Function
port map(RESET,CLK,ES1,open,OA3,Z3);--Z3:output of S3

S4:Sigmoid_Cordic_Sin_Function
port map(RESET,CLK,ES1,open,OA4,Z4);--Z4:output of S4

S5:Sigmoid_Cordic_Sin_Function
port map(RESET,CLK,ES1,open,OA5,Z5);--Z5:output of S5


--Multiplier Mi(i=10~15)

w1k<="100000010101";--w1k=-1.2854211
M11:Multiplier_12bit
port map(RESET,CLK,RQM2,open,Z1,w1k,OM11);--OM11:output of M11

w2k<="100000001111";--w2k=-0.9513310
M12:Multiplier_12bit
```

```
port map(RESET,CLK,RQM2,open,Z2,w2k,OM12);--OM12:output of M12

w3k<="000000001111";--w3k=0.9655813
M13:Multiplier_12bit
port map(RESET,CLK,RQM2,open,Z3,w3k,OM13);--OM13:output of M13

w4k<="000000011011";--w4k=1.7124505
M14:Multiplier_12bit
port map(RESET,CLK,RQM2,open,Z4,w4k,OM14);--OM14:output of M14

w5k<="100000100001";--w5k=-2.0948081
M15:Multiplier_12bit
port map(RESET,CLK,RQM2,open,Z5,w5k,OM15);--OM15:output of M15


--Adder A6

A6:Adder_12bit_input5 port map(OM11,OM12,OM13,OM14,OM15,OA6);--OA6:output of
A6


--Sin function S6

S6:Sigmoid_Cordic_Sin_Function port map(RESET,CLK,ES2,READY,OA6,Zk);


process(CLK,RESET) begin

        if RESET='1' then RQM1<='0';RQM2<='0';ES1<='0';ES2<='0';Q<=0;
        elsif CLK'event and CLK='1' then
                case Q is
                        when 0 => Q<=1;
                        when 1 => RQM1<='1';Q<=Q+1;
                        when 2 to 13 =>Q<=Q+1;
                        when 14 =>ES1<='1';Q<=Q+1;
                        when 15 to 20 =>Q<=Q+1;
                        when 21 =>RQM2<='1';Q<=Q+1;
                        when 22 to 33 =>Q<=Q+1;
                        when 34 =>ES2<='1';Q<=Q+1;
                        when 35 to 41 =>Q<=Q+1;
                        when 42 =>null;
                end case;
        end if;
end process;
end Behavioral;
```