TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI

VIỆN ĐIỆN TỬ - VIỄN THÔNG

BÁO CÁO ĐỒ ÁN THIẾT KẾ III

Đề tài:

TRIỂN KHAI SNN LÊN HỆ THỐNG SOC SỬ DỤNG NỀN TẢNG LITEX

Sinh viên thực hiện:

Họ và tên	: Nguyễn Việt Thi
Mã sinh viên	: 20182798
Lớp	: Điện tử 10- K63

Giảng viên hướng dẫn: TS. HOÀNG PHƯƠNG CHI

Hà Nội, 9-2022

LỜI NÓI ĐẦU

Sự xuất hiện của cuộc cách mạng công nghiệp 4.0 đã mang tới các yêu cầu thay đổi ở quy mô chưa từng có trong cả nền kinh tế thế giới nói chung, cũng như ở Việt Nam nói riêng. Trong công cuộc phát triển đó, ngành điện tử đang đóng vai trò tích cực, quan trọng đối với nền kinh tế xã hội. Bên cạnh đó, một số lĩnh vực công nghệ mới phát triển cũng tác động mạnh tới xu hướng phát triển của nền kinh tế. Ngành trí tuệ nhân tạo và thiết kế vi mạch đang có xu hướng phát triển mạnh mẽ, thay thế nhiều phương pháp thủ công. Việc đó giúp phần cứng có thể xử lý được các tác vụ liên quan đến AI mà tiết kiệm năng lượng, điện tích cũng như tốc độ xử lý cao.

Vì vậy, em chọn đề tài: "**TRIỂN KHAI SNN LÊN HỆ THỐNG SOC SỬ DỤNG NỀN TẢNG LITEX**" . Kiến trúc SNN triển khai trực tiếp trên phần cứng số, mô phỏng cấu trúc và hoạt động của các tế bào thần kinh Neurons trong não người, hướng tới một IC có khả năng xử lý các tác vụ tương ứng với các thuật toán DeepLearning, tuy nhiên lại có sự đột phá về năng lượng tiêu thụ, tốc độ xử lý.

Trong quá trình thực hiện đề tài, vì thời gian và trình độ còn hạn chế nên em sẽ không thể tránh khỏi những sai sót. Em rất mong nhận được những ý kiến đóng góp từ cô để đề tài ngày một hoàn thiện hơn.

Em xin gửi lời cảm ơn chân thành tới cô **Hoàng Phương Chi** đã hướng dẫn và cung cấp những kiến thức cần thiết, tạo điều kiện thuận lợi giúp em hoàn thành đề tài này.

MỤC LỤC

DANH MỤC KÝ HIỆU VÀ CHỮ VIẾT TẮT

Ký hiệu	Ý nghĩa
SNN	Spiking Neural Network
SOC	System on chip
FPGA	Field-programmable gate array
RAM	Random Access Memory
FIFO	First in First Out

DANH MỤC HÌNH VẼ

DANH MỤC BẢNG BIỂU

CHƯƠNG 1. GIỚI THIỆU CHUNG

Chương này trình bày khái quát về ý tưởng của đề tài, mục tiêu và phạm vi cùng như nghiên cứu các phương pháp đã có để thực hiện đề tài.

1.1 Ý tưởng của đề tài

Một hệ thống trên chip đều có ít nhất một vi xử lý làm nhiệm vụ tính toán, và thường thì các hệ thống trên chip hiện nay đều là các hệ thống đa nhân. Các vi xử lý này thường sử dụng các kiến trúc RISC với ưu điểm về việc sử dụng ít tài nguyên tính toán hơn và tiêu thụ ít năng lượng hơn, bởi hạn chế về diện tích chip hay năng lượng cung cấp của các nền tảng tính toán nhúng hay di động khi thời lượng pin là một tiêu chí quan trọng nhưng sự nhỏ gọn cũng cần được ưu tiên. Trong hệ thống trên chip của em thì vi xử lý được sử dụng là VexRiscV, một vi xử lý RISC-V 32-bit có khả năng boot Linux. Vi xử lý này được chọn vì có sự hỗ trợ của hệ sinh thái RISC-V cho phép khởi tạo đơn giản từ các nền tảng xây dựng hệ thống trên chip.

Đồng thời, xu hướng của nền công nghệ trên thế giới hiện nay là phát triển những sản phẩm thông minh, với mục tiêu có thể tự động hóa các hành vi như con người. Tuy nhiên, các thuật toán AI làm tốn quá nhiều tài nguyên và năng lượng và khó ứng dụng vào thực tế.

Là sinh viên ngành Điện tử - Viễn thông, em có nhiệm vụ tìm tòi, nghiên cứu và phát triển các ứng dụng dựa trên cơ sở lý thuyết được học trên trường để có thể tạo ra những ứng dụng trong đời sống thực tiễn hỗ trợ cải thiện cuộc sống của con người. Đề tài này được triển khai với ý tưởng tìm hiểu về cách triển khai trí tuệ nhân tạo trực tiếp trên phần cứng số nhằm tối ưu hóa tốc độ và năng lượng khi triển khai các thuật toán AI.

1.2 Mục tiêu và phạm vi

Mục tiêu chính của đề tài là tìm hiểu về Spiking Neural Network và triển khai lên hệ thống có SOC sử dụng nền tảng LiteX. Đồng thời sử dụng FPGA để kiểm tra chức năng của hệ thống.

1.3 Kết luận

Như vậy, chương mở đầu này đã trình bày khái quát về lý do chọn đề tài, mục tiêu và phạm vi cũng như là nghiên cứu các phương pháp đã có để thực hiện đề tài.

CHƯƠNG 2. CƠ SỞ LÝ THUYẾT

2.1 Giới thiệu về Spiking Neural Network

Mục này sẽ trình bày trình bày tổng quan về Neuronscience Computation cũng như giới thiệu về Spiking Neural Network.

2.1.1 Tổng quan Neuronscience Computation

Neuronscience Computation là mô hình nghiên cứu mô phỏng lại các tế bào thần kinh của não người thực hiện trên phần cứng và thực hiện truyền xử lý thông tin tương tự não người. Neuron science Computation gồm hai hướng chính là Analog Neural Nework (ANN) và Spiking Neural Network (SNN). Tuy nhiên các mạng thần kinh ANN bỏ qua các tế bào thần kinh sinh học trong não và xử lý thông tin dựa trên giá trị số nguyên hoặc số thực. Dựa trên cơ sở đó, vài năm qua đã chứng kiến sự tiến bộ vượt bậc trong việc xây dựng mô hình SNN như một mô hình máy tính mới có khả năng thay thế ANN và là thế hệ tiếp theo của Neural Network.

2.1.2 Giới thiệu Spiking Neural Network

Spiking Neural Networks là một mạng neuron nhân tạo bắt chước mạng neuron tự nhiên. Ngoài tế bào thần kinh và khớp thần kinh, SNNs còn tích hợp thời gian vào mô hình hoạt động của chúng. SNN được coi là thế hệ thứ ba của mạng neuron nhân tạo (ANN)**.** Trong khi ANN cổ điển hoạt động với các đầu vào có giá trị số nguyên hoặc thực, SNN xử lý dữ liệu bằng một chuỗi các spike, về mặt tính toán là các bit 0 và 1.

2.2 Kiến trúc TRUENOTH cho mạng SNN

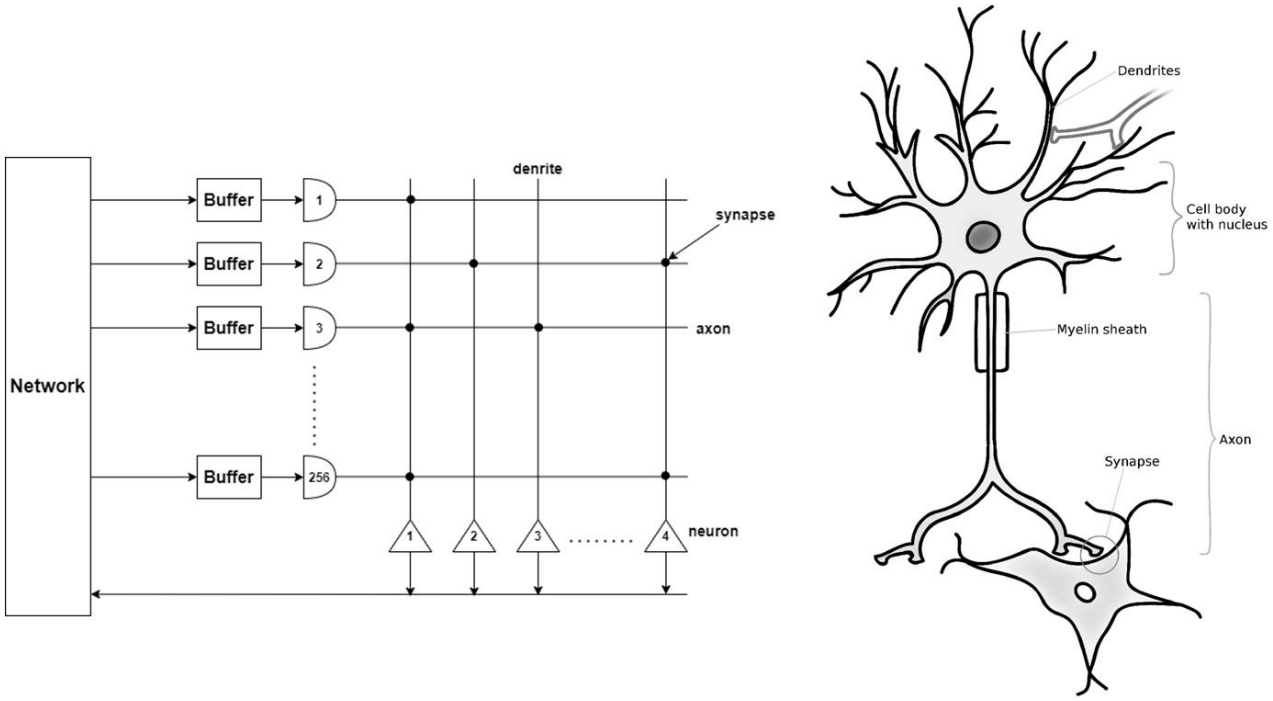
2.2.1 Mô hình tương quan phần cứng - tế bào thần kinh sinh học

Não người được cấu tạo bởi khoảng 3 tỉ neuron kết nối với nhau, một neuron được nối với nhiều neuron khác tạo thành một mạng lưới. Các neuron được nối với nhau thông qua các dây thần kinh gọi là axon, và điểm nối giữa neuron và axon gọi là synapse. Khi ta nhận được một kích thích từ môi trường, ví dụ nhìn thấy ánh sáng, thì ánh sáng đó sẽ được truyền vào mắt mình và mắt mình sẽ chuyển các tín hiệu ánh sáng đó thành tín hiệu điện và truyền vào trong não thông qua dây thần kinh axon từ đó sẽ tới được các dây thần kinh bên trong não. Các neuron bên trong não sẽ tiếp nhận được tín hiệu điện

(spike) đẩy và tích lũy một giá trị gọi là “potential value”, khi giá trị tích lũy vượt quá ngưỡng (threshold) thì nó sẽ phóng ra một kích thích tới con neuron khác mà kết nối với nó.

Mạng SNN cố gắng mô phỏng lại chính xác nhất có thể về cấu tạo và hoạt động của não người, bao gồm gần như đầy đủ các thành phần cơ bản của bộ não, biểu diễn dưới dạng các khối phần cứng số.

Khả năng lưu trữ dữ liệu, hay trí nhớ của bộ não, được hình thành từ việc kết nối các tế bào thần kinh Neurons, dây thần kinh Axons và đặt các trọng số (weights) dựa trên bộ tham số huấn luyện, thay vì có các khối bộ nhớ như RAM, ROM thông thường đối với kiến trúc Von Neumann.



Hình 2.1. Mô hình tương quan các thành phần trong mạng thần kinh

Các thành phần cơ bản của một mô hình mạng lưới thần kinh sinh học:

- Neuron: các tế bào thần kinh tích lũy giá trị màng (Membrain Potential), khi giá trị này vượt một ngưỡng xác định thì Neuron sẽ phóng kích thích.
- Denrite: các đường rẽ nhánh từ một Neuron.
- Axon: các đường dây thần kinh kết nối các Neurons với nhau.
- Synapse: điểm kết nối giữa một Axon và một Denrite.

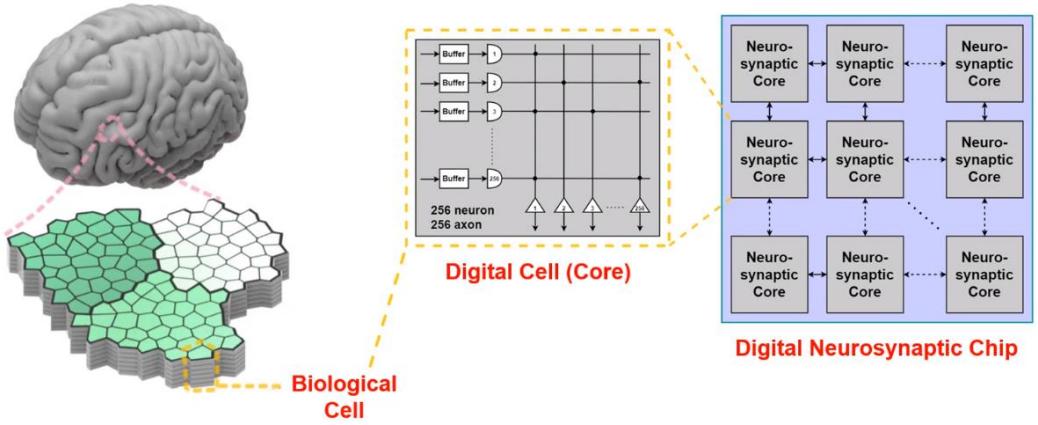
Nhận xét chung:

- Một Neuron có thể nhận được các kích thích từ nhiều Axons thông qua các Denrites của nó.
- Một Axons có thể kết nối tới nhiều Neuron để lan truyền kích thích.
- (Kết nối Synapse giữa Axon và Denrite được thể hiện bằng chấm đậm)

2.2.2 Mô hình mạng lưới tế bào chia thành nhiều Core

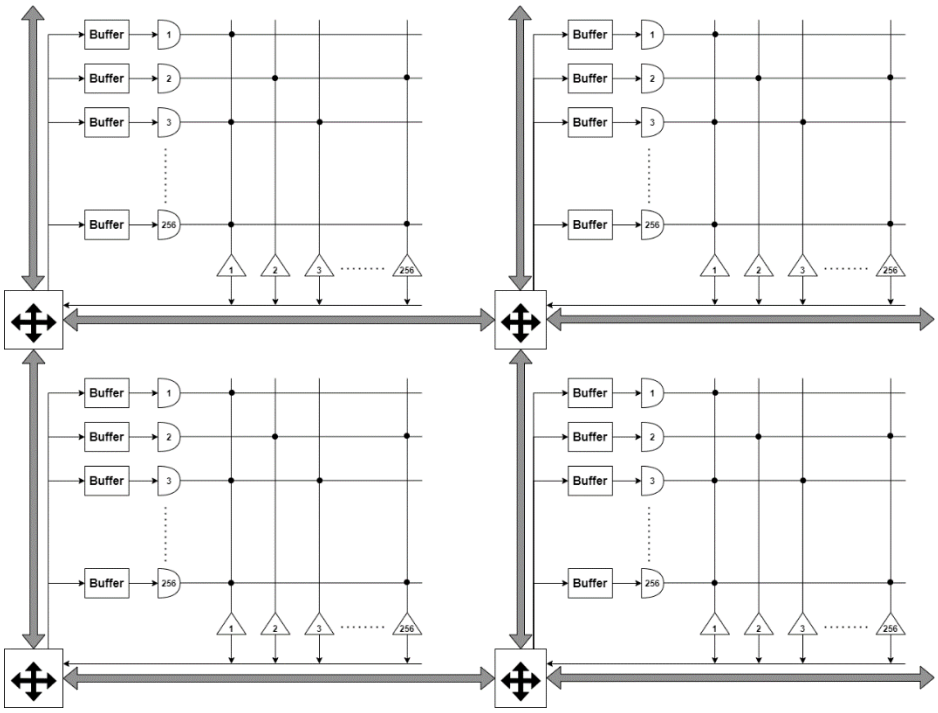
Ta thực hiện cách mô hình não người bằng cách sử dụng các phần tử điện tử và chia các neuron thành rất nhiều các khối nhỏ, mỗi khối nhỏ là một core.

Mỗi Core mô phỏng lại một phần sinh học của bộ não (cell), bao gồm 256 Neurons, nhận kích thích (spikes) từ mạng, tích lũy giá trị Potential dựa trên các kích thích này và có thể phóng ra các kích thích tương ứng nếu giá trị Potential vượt qua một ngưỡng nhất định (threshold). Sau khi phóng ra một kích thích, giá trị Potential của Neuron sẽ được thiết lập lại về giá trị khởi tạo. Còn việc kết nối neuron nào với neuron nào sẽ thu được qua quá trình training và cuối cùng ta sẽ có một mạng gồm rất nhiều các core như sau:



Hình 2.2 Kiến trúc TrueNorth cho mạng SNN [1]

Khi hết hợp nhiều Core thành một mạng lưới lớn, mạng SNN được hình thành và thực hiện được các chức năng ứng với các bộ tham số kết nối đã được huấn luyện từ trước. Khi số lượng các Core tăng lên, mạng càng thực hiện được các chức năng phức tạp hơn, với độ chính xác cao hơn, mang tính ứng dụng thực tiễn.

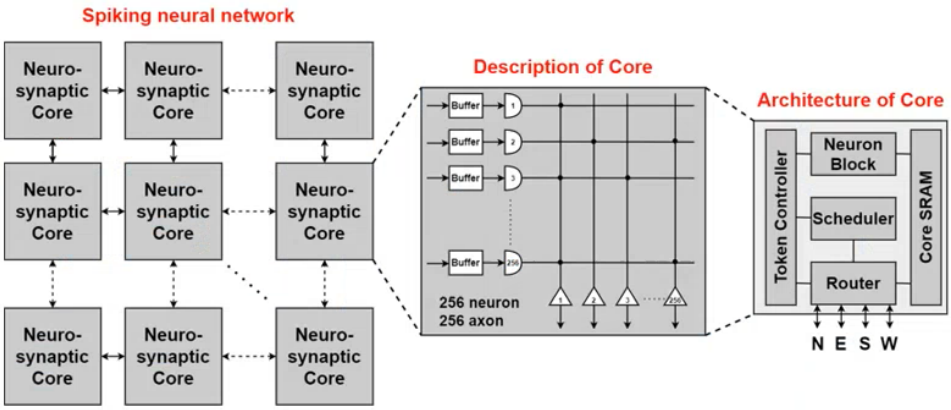


Hình 2.3 Mạng SNN hình thành từ các Core

Kích thích phóng ra từ Neurons trong mỗi Core sẽ được đưa tới các Neurons khác trong mạng bằng khối Router trong Core đó, thông qua các dây thần kinh Axons. Mỗi Neuron đều chỉ phóng kích thích tới một đích đến duy nhất, xác định bởi các trường trong một gói tin được đề cập tại mục 2.2.4.

2.2.3 Kiến trúc của một Core (Cell) trong mạng

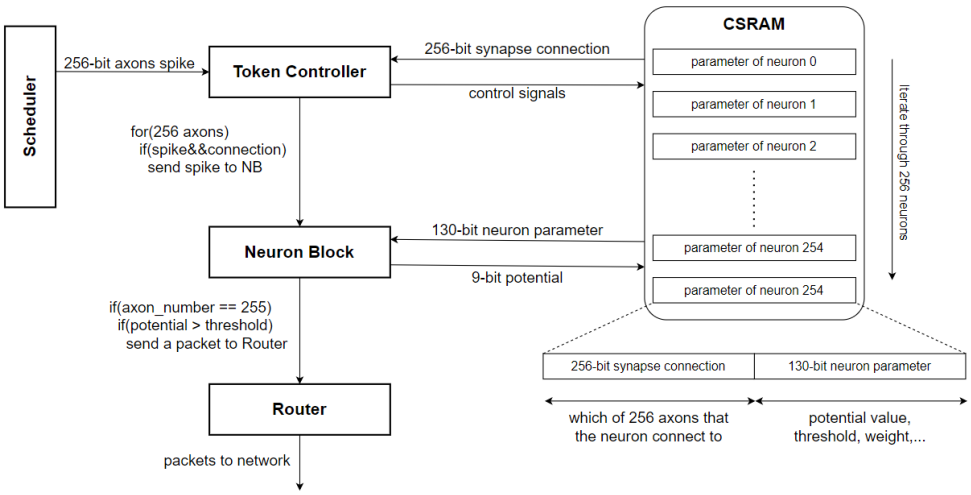
Kiến trúc True North được đề xuất với mỗi Core gồm 5 khối chính bao gồm Neuron Block, CSRAM, Scheduler, Token Controller và Router.



Hình 2.4 Kiến trúc của một Core trong mạng [1]

Mạng SNN khi này có thể mở rộng không giới hạn dựa trên các Core khi kết nối với nhau, số lượng Core càng lớn thì khả năng ứng dụng của mạng càng cao, thực hiện

được nhiều chức năng phức tạp hơn, tuy nhiên diện tích mạch và năng lượng tiêu thụ sẽ lớn hơn.



Hình 2.5 Giao tiếp giữa các khối trong một Core

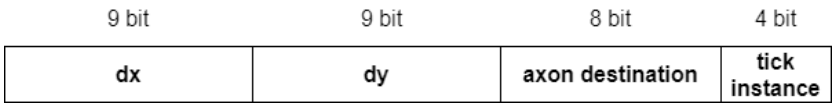
Hình 2.5 mô tả giao tiếp giữa các khối trong một Core của kiến trúc Truenorth.

Khi Router nhận được gói tin có đích đến là Core hiện tại, nó sẽ đưa các gói kích thích tới Scheduler để lưu trữ. Tại tick tiếp theo, các gói này quyết định việc những Axons nào trong 256 Axons của Core sẽ phóng ra kích thích. Sau đó Token Controller là một máy trạng thái sẽ điều khiển hoạt động của CSRAM và Neuron Block một cách phù hợp để tính toán đầu ra cho 256 Neurons, đưa các kích thích tới mạng qua Router.

Lưu ý: ngoài chu kỳ đồng hồ clk, kiến trúc mạng sử dụng thêm một đơn vị tính toán thời gian là “tick”. Mỗi tick sẽ được kích hoạt sau một số chu kỳ clk nhất định, bắt đầu cho toàn bộ hoạt động tính toán của một Core. Nói cách khác, khoảng thời gian giữa các tick là thời gian mà mạng cần để xử lý hoàn tất cho một ảnh. Việc lựa chọn giá trị tick hợp lý cũng quyết định rất nhiều tới tốc độ xử lý của mạng. Nếu thời gian tick quá nhỏ, ảnh chưa kịp xử lý xong đã xuất hiện tick mới, gây ra lỗi cho mạng. Ngược lại nếu thời gian tick quá lớn, mạng xử lý xong ảnh sẽ phải chờ tick mới xuất hiện để tiếp tục hoạt động, gây lãng phí hiệu năng của mạng.

2.2.4 Cấu trúc của mỗi gói ứng với một kích thích lan truyền trong mạng

Mỗi gói kích thích lan truyền trong mạng được biểu diễn bởi một bus dữ liệu có kích thước 30 bit, gồm các trường dx, dy, axon destination và tick instance.



Hình 2.6 Cấu trúc một gói kích thích

Trong đó, ý nghĩa các trường được thể hiện:

- dx, dy: Hai trường ứng với chiều ngang, dọc, xác định vị trí của Core đích mà gói tin hướng đến. Ví dụ khi dx = 1 và dy = 1 thì đầu tiên, gói tin sẽ đi sang phải 1 bước, sau đó sẽ trừ dx cho 1, lúc này dx sẽ còn giá trị là 0, nghĩa là đã đi đủ bước sang phải, tiếp tục kiểm tra dy thấy dy = 1 nghĩa là phải đi lên trên 1 bước, còn nếu dy = -1 thì nó đi xuống dưới 1 bước. Khi đến đúng Core đích thì cả dx và dy đều có giá trị là 0.
- Axon destination: Sau khi gói tin đã đến được Core mong muốn thì Router sẽ đưa gói tin vào Scheduler và nó sẽ được gửi vào 1 trong số 256 Axon, để biết được nó sẽ đưa vào Axon nào thì trường Axon destination chỉ ra vị trí Axon mà gói tin gửi tới đó sẽ nhận được kích thích để đưa tới các Neurons trong Core. Ví dụ Axon destination có giá trị là 2, nó sẽ được đưa vào Axon có vị trí thứ 2
- Tick instance: Sau khi gói được đưa từ Router tới Scheduler để lưu trữ, trường tick instance xác định thời điểm mà kích thích được đưa ra từ Scheduler tới các Axons trong Core. Giá trị tick instance càng lớn thì kích thích càng lâu được đưa ra hơn, và khoảng thời gian này là số nguyên lần của thời gian "tick".

Trong phần code, chúng em để dữ liệu các trường trên là kiểu int.

Lưu ý: hai khái niệm "tick" và "tick instance" là hoàn toàn khác nhau. Trong đó "tick" là một tín hiệu được kích hoạt định kỳ sau mỗi khoảng thời gian nhất định, được dùng để xác định thời gian xử lý xong một ảnh. Còn "tick instance" mang ý nghĩa là một trường trong gói kích thích lan truyền trong mạng.

2.2.5 Kiến trúc và hoạt động của các khối con trong một Core

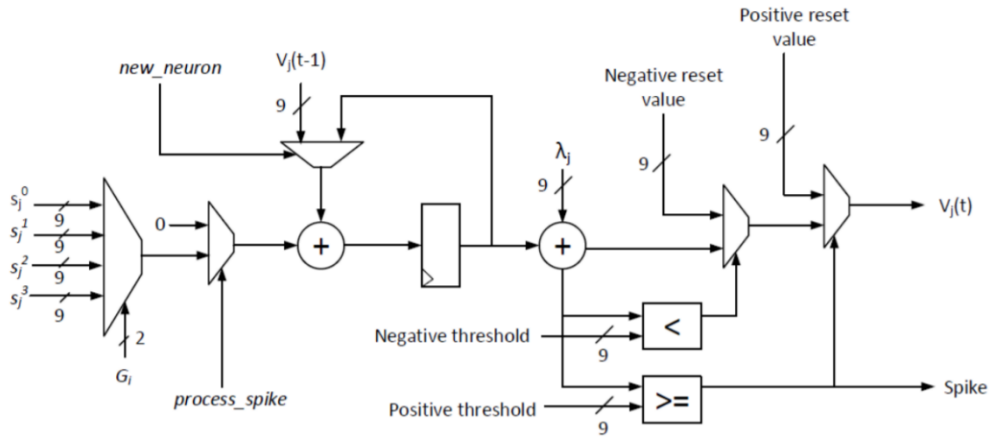
2.2.5.1 Neuron Block

Neuron Block là khối tính toán chính trong kiến trúc TrueNorth. Các bước tính toán xoay quanh một giá trị gọi là Potential (Membrain Potential), là giá trị tích lũy của Neuron, và giá trị này hoàn toàn độc lập giữa các Neuron khác nhau. Trong một Core

gồm 256 Neuron (là những hình tam giác ở Hình 2.3), các neuron này sẽ tích lũy giá trị và bắn các spike vào mạng (mạng chung thực chất là gồm nhiều các core khác).

Các neuron trong mạng bắn kích thích vào đường axon nằm ngang và địa chỉ của chúng dựa vào trường "axon destination" của spike được gửi. Khi một axon nhận được kích thích từ trong mạng bắn vào, nó sẽ kiểm tra xem những con neuron nào có kết nối với cả dây thần kinh đấy (là dấu chấm đen ở Hình 2.3) là các kết nối synapse, khi phát hiện có kết nối nó sẽ gửi giá trị đấy vào neuron. Chỉ cần đủ 2 điều kiện là có một kích thích được bắn vào dây thần kinh, và dây thần kinh đấy được nối với một con neuron khác thì con neuron đó sẽ nhận được một kích thích.

Bản thân khối Neuron Block trong thiết kế chỉ là một "lớp vỏ" gồm các khối cộng trừ, so sánh. Các thuộc tính của Neuron được biểu diễn bằng một bộ tham số gồm 368 bit và các tham số này đều được lưu trữ trong CSRAM, ý nghĩa các trường trong bộ tham số sẽ được nói rõ hơn ở mục sau.



Hình 2.7 Kiến trúc giản lược Neuron Block [2]

Tại một thời điểm xác định sẽ chỉ thật sự tồn tại một Neuron đang hoạt động thay vì 256 Neurons trong 1 Core. Tức là, Neuron Block sẽ lần lượt được nạp những bộ tham số của các Neuron từ 0 đến 255, sau khi tính toán xong cho một Neuron, CSRAM lại tiếp tục nạp Neuron tiếp theo vào Neuron Block để tính toán. Quá trình "nạp nhân" vào "lớp vỏ" này được điều khiển bởi Token Controller trong mục 2.2.5.5.

Khi một kích thích được đưa vào Neuron, một trong số 4 trọng số được chọn dựa trên bộ tham số, giá trị này được cộng tích lũy với giá trị Potential ở trạng thái hiện tại của Neuron, tạo nên giá trị Potential ở thời điểm tiếp theo cho Neuron. Giá trị Potential khi này được so sánh với một giá trị ngưỡng (gồm ngưỡng âm và ngưỡng dương). Nếu

giá trị Potential vượt quá ngưỡng dương, Neuron sẽ phóng ra một kích thích và thiết lập lại giá trị Potential về giá trị khởi tạo dương (Positive reset value). Ngược lại, nếu vượt ngưỡng âm, Neuron được thiết lập về giá trị khởi tạo âm (Negative reset value) mà không phóng ra kích thích. Nếu giá trị Potential không vượt qua ngưỡng âm hoặc ngưỡng dương, kích thích cũng sẽ không được phóng ra mạng, giá trị Potential của Neuron được bảo toàn cho lần tính toán tiếp theo. Các giá trị về ngưỡng hay giá trị reset đều được lưu trong CSRAM.

2.2.5.2 CSRAM

CSRAM là khối lưu trữ chính của một Core, là một bộ nhớ gồm 256 hàng, trong đó mỗi hàng là 1 bộ 368 bits tham số ứng với một Neuron. Tất cả tham số của mỗi neuron (weights, liên kết, giá trị potential, giá trị reset, ngưỡng, vị trí đến của spike được gửi đi) được lưu trong CSRAM. Bộ tham số này thu được sau quá trình huấn luyện cho mạng. Các trường trong 368 bits tham số được thể hiện trong bảng sau:

Bảng 2.1. Bảng mô tả các trường trong CSRAM

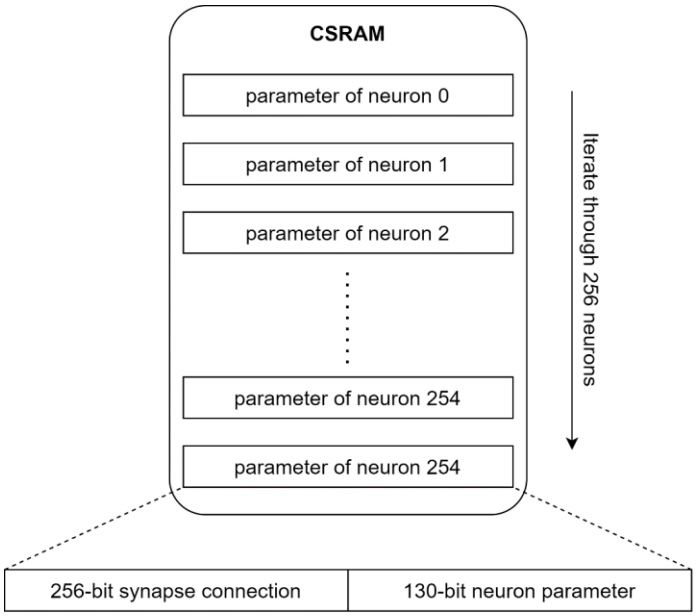
Parameter	Width	Description
Synapse Connections	256 bits	Thể hiện kết nối của Neuron đang xét tới 256 Axons trong Core, bit 1 ứng với có kết nối.
Current Potential	9 bits	Giá trị Potential tích lũy của Neuron.
Reset Potential	9 bits	Giá trị Potential của Neuron sẽ được reset về giá trị này khi Neuron phóng ra kích thích.
Weights 0 to 3	9 bits each	Giá trị trọng số của kết nối synapse đặc trưng cho từng Neuron,
Leak Value	9 bits	Giá trị Potential rò rỉ khi tính toán.
Positive Threshold	9 bits	Ngưỡng Potential dương.
Negative Threshold	9 bits	Ngưỡng Potential âm.
Reset mode	1 bit	Lựa chọn 2 Mode reset khác nhau của Neuron.
Spike Destination	26 bits	Gồm các trường dx, dy rộng 9 bit và Axons destination 8 bit, giống với cấu trúc gói tin

Tick delivery	4 bits	Giống với tick instance trong gói tin

Các tham số này quyết định tới hoạt động tính toán của Neuron, cũng như đích đến của các kích thích sinh ra bởi neuron.

Hoạt động của CSRAM được điều khiển bởi khối Token Controller. Khi bắt đầu mỗi tick, các tham số về neuron chứa trong CSRAM được nhập vào Neuron Block. Sau khi hoàn thành tính toán với một neuron, 9 bits Current Potential được cập nhật, sửa đổi và ghi lại vào CSRAM. Giá trị này ứng với sự thay đổi Potential của Neuron. Token Controller sẽ thực hiện trên bộ tham số của neuron tiếp theo hoặc quay về trạng thái Idle để chờ tick tiếp theo.

Hình 2.8 mô tả cấu trúc cơ bản của khối CSRAM. Trong đó, mỗi phần tử của khối là 1 bộ tham số của Neuron, với 256 bits đầu là giá trị liên kết của Neuron đang xét với các axon trong mạng, 130 bits sau là các tham số weights, potential, threshold, …. Token Controller sẽ thực hiện duyệt qua từng bộ tham số của các Neuron theo thứ tự từ 0-256.

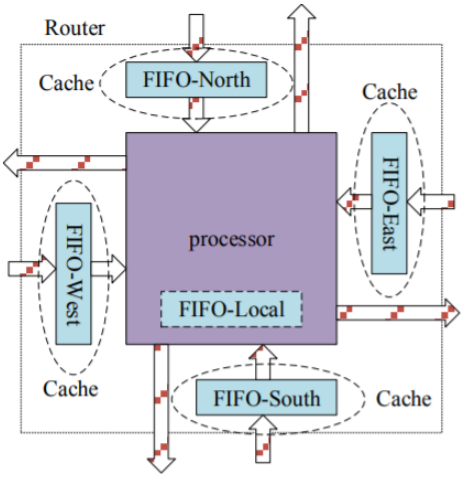


Hình 2.8 Cấu trúc khối CSRAM

2.2.5.3 Router

Trong quá trình tính toán của một Neuron, Router là thành phần đảm bảo gói kích thích được tạo ra bởi các Neuron trong Core đến được vị trí mong muốn, hay nói cách khác, Router phụ trách nhiệm vụ phân phối các gói kích thích lan truyền giữa các Core trong mạng.

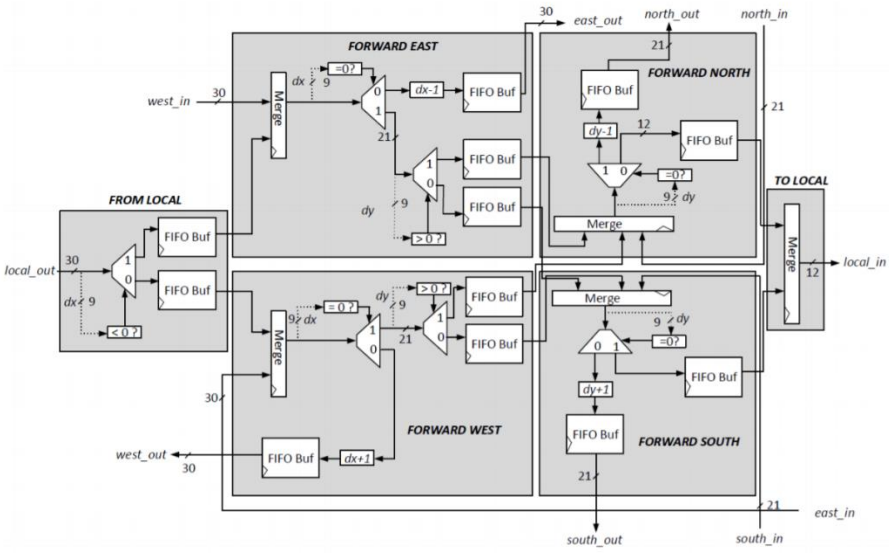
Để mô tả một cách đơn giản hoạt động của Router, chúng em dựa vào 5 hướng mà nó có thể nhận hoặc gửi các gói kích thích: Đông, Tây, Nam, Bắc và từ chính Core hiện tại. Kích thích nhận được từ các hướng được lưu tại FIFO tại hướng đó, sau đó tiến hành xử lý bằng các thành phần logic để gửi đến vị trí đích mong muốn. FIFO có kích thước càng lớn thì Router nhận và lưu trữ được càng nhiều gói kích thích. Hình 2.9 là sơ đồ khối chức năng, mô tả đơn giản cách hoạt động của Router.



Hình 2.9 Sơ đồ khối chức năng Router

Để chuyển các gói kích thích nhận được đến vị trí đích, cần phải có các thành phần đóng vai trò xử lý tham số chứa trong gói tin. Khi một gói kích thích được đưa vào Router, nó sẽ bóc tách các trường trong gói tin để tính toán và quyết định việc chuyển tiếp gói tin này sang Core nào trong 4 Core lân cận theo các hướng Đông, Tây, Nam, Bắc hay lưu tại Core đang xét. Nếu gói kích thích có địa chỉ đích chính là Core đang xét, gói sẽ được Router đưa vào trong Scheduler của Core thay vì đưa sang các Core xung quanh. Dựa vào đó, ta xây dựng được cấu trúc thành phần logic được mô tả như trong Hình 2.10.

Router gồm 6 khối chính là From Local, To Local, Forward East, Forward West, Forward North, Forward South có chức năng lưu trữ và xử lý gói tin như mô tả phía trên.

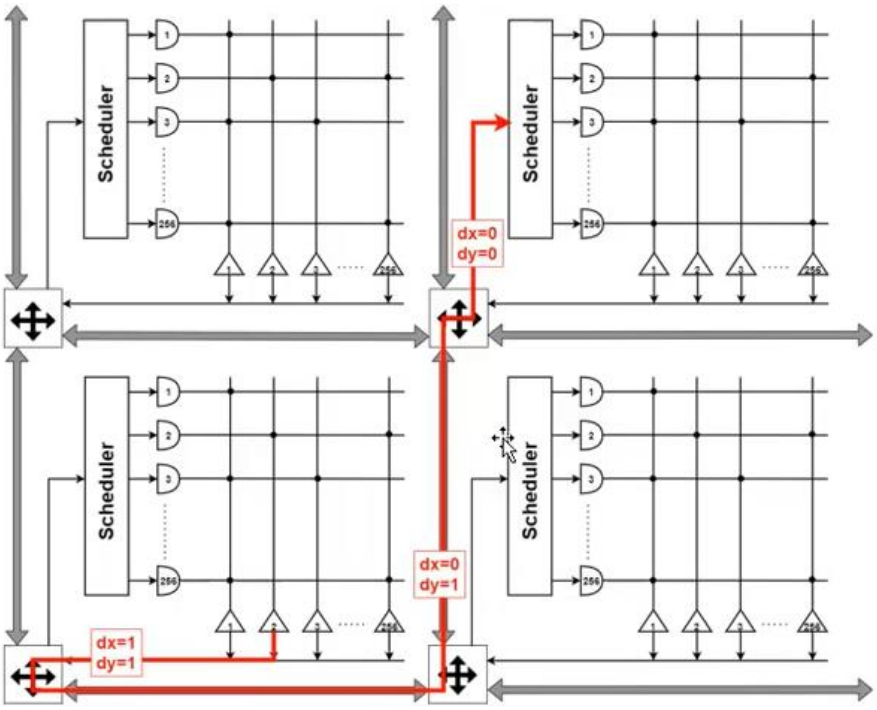


Hình 2.10 Kiến trúc Router [1]

Một gói kích thích có thể đi tới Router từ các Core lân cận ở 4 hướng Đông, Tây, Nam, Bắc ứng với các đường east_in, west_in, south_in và north_in. Ngoài ra, cũng có thể gói tin tới Router được sinh ra bởi chính các Neurons trong Core, khi các Neurons này thực hiện tính toán và đưa ra các kích thích tới Router thông qua khối From Local .

Một gói kích thích đưa ra từ Router có thể được chuyển sang các Core lân cận theo 4 hướng Đông, Tây, Nam, Bắc thông qua các khối Forward East, Forward West, Forward South và Forward North. Nếu gói tin có đích đến là Core đang xét, kích thích sẽ được đưa vào Core thông qua khối To Local. Khi này thì gói kích thích chỉ còn giữ lại 12 bit gồm tick instance và axon destination.

Các đường màu đỏ trong Hình 2.11 mô tả các ví dụ về việc gói kích thích được phóng ra từ một Neuron trong Core, sau đó được gửi tới một đích đến xác định thông qua các Router trên đường di chuyển. Các gói sẽ được di chuyển lần lượt theo chiều ngang, dọc và tới được Core đích, cũng như Axon đích tại Core này.



Hình 2.11 Gói kích thích được lan truyền thông qua các Router

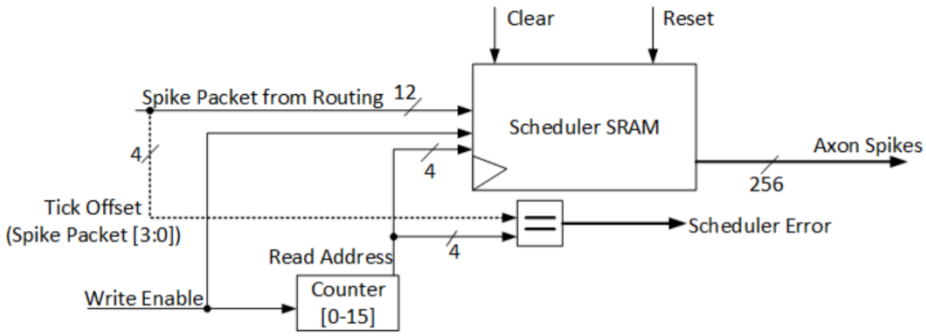
Một gói kích thích có cấu trúc như trong Hình 2.6 được lan truyền trong mạng lưới gồm nhiều Core như sau:

- Đầu tiên, Router trích xuất trường dx trong gói, đưa gói kích thích di chuyển theo chiều ngang:
  - → Nếu dx > 0: gói kích thích được truyền sang Core hướng Đông (sang phải), thay đổi trường dx trong gói: dx = dx – 1.
  - → Nếu dx < 0: gói kích thích được truyền sang Core hướng Tây (sang trái), thay đổi trường dx trong gói: dx = dx + 1.
  - → Nếu dx = 0: loại bỏ trường dx trong gói, bắt đầu truyền dọc.
- Sau khi giá trị dx = 0, gói kích thích loại bỏ đi trường dx, khi này gói chỉ còn 21 bits và tiếp tục được di chuyển theo chiều dọc dựa theo trường dy:
  - → Nếu dy > 0: gói kích thích được truyền sang Core hướng Bắc (lên trên), thay đổi trường dy trong gói: dy = dy – 1.
  - → Nếu dy < 0: gói kích thích được truyền sang Core hướng Nam (xuống dưới), thay đổi trường dy trong gói: dy = dy + 1.
  - → Nếu dy = 0: gói kích thích đã tới được Core đích. Tiếp tục loại bỏ trường dy trong gói và đưa gói vào Scheduler trong Core đang xét.

2.2.5.4 Scheduler

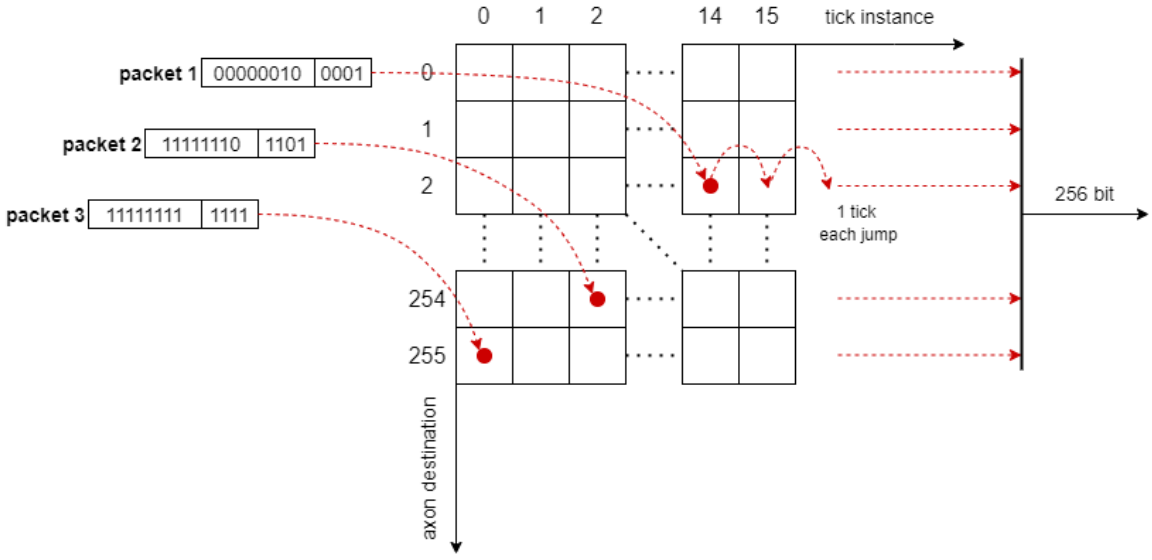
Khi các gói kích thích truyền thành công tới Core đích, các gói này được Router trong Core này đưa vào lưu trữ tại Scheduler. Khi này gói tin chỉ còn 12 bits gồm hai trường tick instance và axon destination, do các trường xác định Core là dx, dy đã bị loại bỏ.

Tại tick tiếp theo, Scheduler thực hiện nhiệm vụ phân phát các gói kích thích đã được nhận từ Router trong các tick trước đó tới 256 Axons trong Core. Như vậy Scheduler cần một khối "Scheduler SRAM" để lưu trữ, cùng với đó là một khối sinh địa chỉ đọc, ứng với bộ đếm Counter, thể hiện như trong Hình 2.12.



Hình 2.12 Kiến trúc Scheduler [2]

Trong đó khối Scheduler SRAM có cấu trúc gồm 256 hàng và 16 cột, ứng với hai trường là axon destination 8 bits và tick instance 4 bits.



Hình 2.13 Mô tả hoạt động của Scheduler

Hoạt động của Scheduler được mô tả một cách đơn giản hơn như sau (ứng với Hình 2.13):

- Tại mỗi tick, các gói được gửi tới Scheduler và điền vào các ô nhớ dưới dạng các chấm đỏ như trong hình. Việc điền vào hàng nào được xác định bởi 8 bits axon destination, ứng với việc kích thích đó sẽ được phóng ra từ Axon nào. Tương tự, việc điền vào cột nào được xác định bởi 4 bits tick instance, xác định thời điểm mà Axon trong core sẽ phóng ra kích thích.
- Sau mỗi tick, dữ liệu trong các cột đều được dịch phải sang cột lân cận, các gói trong cột thứ cuối cùng (cột 15) được đưa ra thành các kích thích của 256 Axons trong tick hiện tại, ứng với 256 bits Axon Spikes trong Hình 2.13.

Như vậy, có thể coi Scheduler thực hiện 2 chức năng chính như sau:

- Với 4 bits tick instance: Scheduler có chức năng như một hàng đợi/tập thanh ghi dịch. Các gói kích thích sẽ được giữ tại Scheduler tối đa trong khoảng thời gian 16 ticks, trước khi chuyển thành các kích thích phóng tới các Neuron.
- Với 8 bits axon destination: Scheduler có chức năng như một bộ giải mã. Đầu vào là các gói với 8 bit axon destination dưới dạng mã nhị phân, đầu ra là 256 bits Axon Spikes (các vị trí có bit 1 trong 256 bits Axon Spikes thể hiện là có kích thích và ngược lại với bit 0).

2.2.5.5 Token Controller

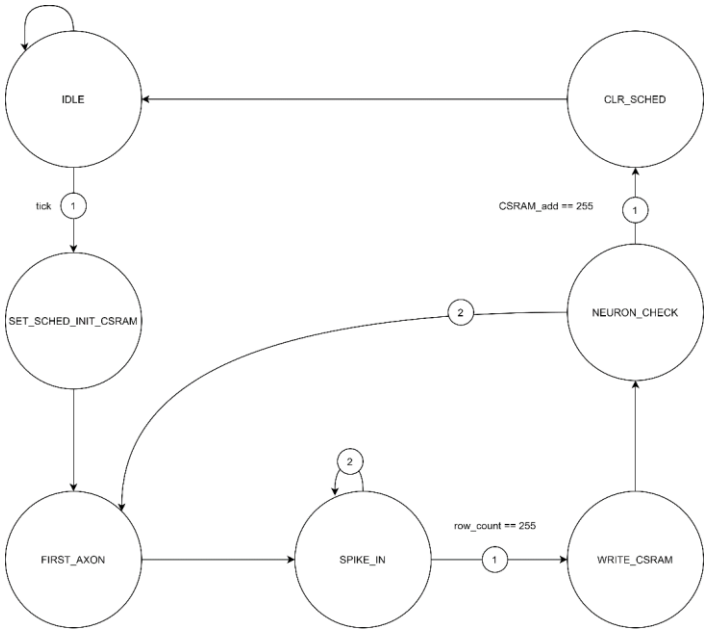
Token Controller là một máy trạng thái với tổng cộng 6 trạng thái, điều khiển quá trình "nạp nhân" lần lượt cho 256 Neurons từ CSRAM vào Neuron Block (Hình 2.14).

Các trạng thái được mô tả như sau:

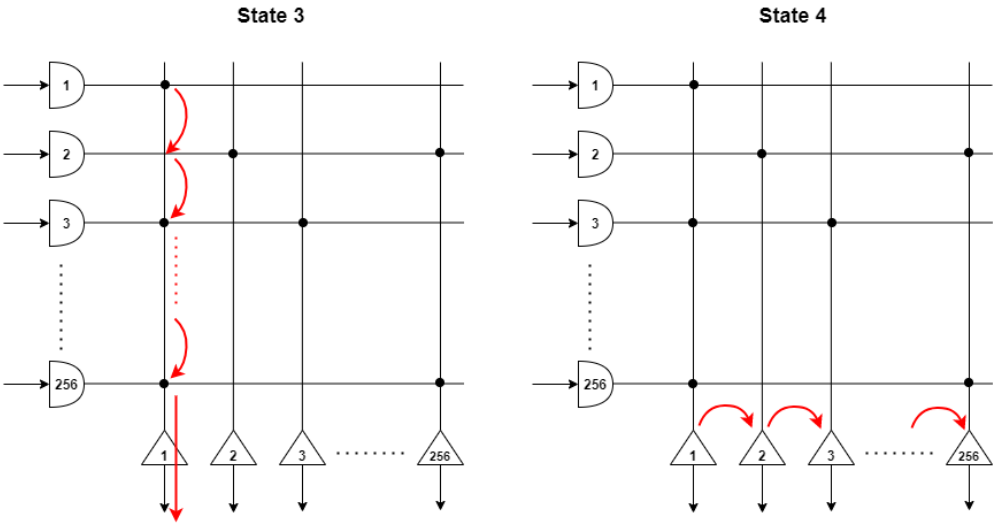
- IDLE: Trạng thái khởi đầu, chờ tính hiệu "tick" để chuyển sang trạng thái tiếp theo.
- SET_SCHED_INIT_CSRAM: Gửi tín hiệu đến bộ Scheduler và khởi tạo địa chỉ của CSRAM đến đọc dữ liệu tử Neuron đầu tiên (CSRAM_addr = 0). Tín hiệu đến bộ Scheduler (set) tăng bộ đếm địa chỉ để lấy đúng "axon spike" cho tick sau.
- FIRST_AXON: Đây là trạng thái đầu tiên trong số hai trạng thái mà chúng tôi có thể xử lý spike, so sánh axon spike[0] với synapse [0] (được lấy từ CSRAM với vị trí của neuron trong CSRAM là CSRAM_addr) nếu bằng thì "current potential" lưu trong thanh ghi của neuron block tăng lên 1 lượng "Weights" (Giá

trị trọng số của kết nối synapse đặc trưng cho từng Neuron) còn không "current potential" sẽ giữ nguyên.

- SPIKE_IN: Tăng "row_count" và so sánh "axon spike[row_count]" với "synapse [row_count]" và đưa các tín hiệu vào khối Neural Block để xử lý. Nếu row_count bằng số "Neuron – 1" thì sẽ chuyển lên trạng thái trên, sai thì sẽ lặp lại trạng thái đó.
- WRITE_CSRAM: Sau khi tính toán xong 256 Axons cho một Neuron, nếu Neuron này đạt tới giá trị ngưỡng và phóng ra kích thích, Token Controller cũng nhận kích thích này và gửi tín hiệu đến khối Router, đồng thời cập nhật lại giá trị "Current Potential" của Neuron đang xét vào trong CSRAM.
- NEURON_CHECK: Ở trạng thái này, đã duyệt qua tất cả các "axon spike" và so sánh với "synapse[CSRAM_addr]". Đồng thời tăng "CSRAM_addr", nếu "CSRAM_addr" không vượt qua số Neuron thì sẽ chuyển về trạng thái FIRST_AXON cón không sẽ đưa lên trạng thái CLR_SCHED.
- CLR_SCHED: Gửi tín hiệu đến bộ scheduler sẽ xóa toàn bộ "axon spikes" mới được xử lý (xóa toàn bộ dữ liệu trong cột cuối cùng của khối Scheduler). Sau đó quay trở lại trạng thái IDLE đến khi nhận được "tick".



Hình 2.14 Máy trạng thái Token Controller



Hình 2.15 Mô tả hoạt động Token Controller

Hình 2.15 mô tả quá trình tính toán của một Neuron khi duyệt qua 256 Axons theo chiều dọc, và quá trình này được lặp lại cho toàn bộ 256 Neurons trong Core.

2.3 Cơ sở lý thuyết về hệ sinh thái RISC-V

Hệ sinh thái RISC-V bao gồm các lõi vi xử lý RISC-V, các hệ thống trên chip sử dụng vi xử lý RISC-V và các phần mềm hỗ trợ việc phát triển, kiểm tra, mô phỏng các hệ thống trên. Để có cái nhìn tổng quan về sự phát triển của hệ sinh thái RISC-V ta cần nắm được thông tin về sự phát triển của từng thành phần trong hệ sinh thái. Một số nội dung về sự phát triển của hệ sinh thái RISC-V liên tục được cập nhật trên trang chủ của cộng đồng.

Về các vi xử lý RICS-V, trong thời gian từ 2016-2022, số lượng các vi xử lý được nghiên cứu mới ngày một nhiều. Trong đó có các vi xử lý được nghiên cứu bởi các trường đại học để phục vụ cho các mục đích giảng dạy. Cũng có các vi xử lý được nghiên cứu ở các trường đại học với mục đích tạo ra các vi xử lý mã nguồn mở nhưng có thể cạnh tranh với các vi xử lý thương mại trên thị trường. Và cả những vi xử lý được tạo ra bởi các công ty thương mại với mục đích tham gia vào thị trường thiết kế vi xử lý, hay phục vụ cho các sản phẩm khác của công ty, thí dụ như vi xử lý cho các thiết bị điện tử thương mại của công ty.

Các vi xử lý RISC-V được phát triển bằng nhiều loại ngôn ngữ khác nhau, sử dụng nhiều nền tảng khác nhau. Tuy nhiên để có thể tích hợp các lõi vi xử lý với các thiết bị ngoại vi và các bus giao tiếp để tạo thành một hệ thống tính toán hoàn chỉnh, thì cần sự

hỗ trợ của các nền tảng phần mềm. Trong đó có thể kể đến các nền tảng ChipYard, PULP và LiteX.

Về các phần mềm liên quan đến nền tảng RISC-V thì ta có nhiều loại phần mềm khác nhau, từ các phần mềm hỗ trợ việc mô phỏng hoạt động của các hệ thống hay vi xử lý phục vụ cho các mục đích từ giáo dục cho đến thiết kế, đến các phần mềm hỗ trợ việc xây dựng một môi trường phần mềm trên hệ sinh thái RISC-V như các hệ điều hành Unix được xây dựng riêng cho vi xử lý RISC-V, các thư viện và trình biên dịch C để chạy phần mềm bare-metal trên RISC-V hay các trình gỡ lỗi, trình kiểm thử phục vụ cho quá trình phát triển các vi xử lý/ hệ thống chip RISC-V, cho đến các hỗ trợ cho việc xây dựng các phần mềm ứng dụng chạy trên nền tảng hệ điều hành và nền tảng phần mềm RISC-V như các thư viện về học máy và trí tuệ nhân tạo, hay như các phần mềm bảo mật cho các nền tảng RISC-V. Sự phát triển của phần mềm phục vụ nền tảng RISC-V được thể hiện qua sự tăng trưởng không ngừng từ số lượng các phân loại chức năng của các phần mềm để tạo nên hệ sinh thái phần mềm ngày một đa dạng mà còn từ số lượng phần mềm trong mỗi phân loại, thể hiện sự quan tâm của cộng đồng mã mở cho hệ sinh thái phần mềm RISC-V ngày một tăng cao. Chính sự phát triển này thể hiện được triển vọng phát triển của hệ sinh thái RISC-V vì đa số các nền tảng phần cứng mã mở non trẻ được tạo ra để cạnh tranh với các nền tảng thương mại đã tồn tại nhiều năm với hệ sinh thái đã phát triển cực kì hoàn thiện, nên rất khó thể hiện được tiềm năng của mình và rất khó thu hút các kĩ sư phần mềm để hoàn thiện một hệ sinh thái đầy đủ kết hợp giữa phần cứng và phần mềm do không có nguồn vốn. Sự thành công của hệ sinh thái RISC-V có sự đóng góp rất lớn của các phần mềm để hoàn thiện hệ sinh thái đó.

2.4 Kiến trúc tập lệnh RISC-V

Kiến trúc tập lệnh RISC-V được ra đời nhằm cạnh tranh với các kiến trúc tập lệnh phổ biến hiện tại như ARM. Nhằm phục vụ cho quá trình phát triển và tầm nhìn lâu dài, ngay từ đầu kiến trúc tập lệnh RISC-V được chia ra làm các tập lệnh cơ bản và các tập lệnh mở rộng. Các tập lệnh cơ bản chứa các lệnh cần thiết để tạo ra một vi xử lý hoạt động được. Các tập lệnh mở rộng chứa các lệnh có cùng nhóm chức năng, nhằm mở rộng khả năng hoạt động của các vi xử lý một cách linh hoạt nhất. Khi thiết kế một vi xử lý RISC-V, ta có thể xác định trước nhu cầu tính toán cần thiết hay các chức năng cần hỗ trợ để có thể chọn các tập lệnh được hỗ trợ cho đúng và đủ nhằm tối ưu hiệu năng và độ phức tạp của vi xử lý.

Kiến trúc tập lệnh cơ bản của RISC-V là các lệnh về số nguyên với các kích cỡ lệnh là 32-bit, 64-bit hoặc 128-bit nhưng hỗ trợ các không gian địa chỉ bộ nhớ khác nhau. Các tập lệnh này được gọi tên lần lượt là RV32I, RV64I và RV128I. Để triển khai các tập lệnh này thì các thanh ghi cần có kích cơ tương ứng: RV32I yêu cầu các thanh ghi 32-bit, RV64I yêu cầu các thanh ghi 64-bit và tương tự cho RV128I. Việc phân chia này nhằm giúp tập lệnh cơ bản có thể được lựa chọn phù hợp cho các dòng vi xử lý khác nhau. Trong đó RV32I phù hợp với các vi điều khiển có kích cỡ nhỏ nhằm tiết kiệm năng lượng, còn RV64I phù hợp với các vi xử lý có nhu cầu cao hơn về hiệu năng, do thường đi kèm với nhiều tập lệnh mở rộng hơn cũng như không gian địa chỉ bộ nhớ lớn hơn nên hỗ trợ được dung lượng bộ nhớ lớn. Trong tương lai có thể sẽ có lúc các ứng dụng yêu cầu không gian địa chỉ nhớ 128-bit nên RISC-V có hỗ trợ cho RV128I, nhưng chủ yếu trong giai đoạn hiện tại các vi xử lý vẫn chỉ tích hợp các tập lệnh 32-bit hoặc 64-bit.

Trong phạm vi chương này ta sẽ tìm hiểu về kiến trúc tập lệnh RV32I do sự khác biệt chủ yếu giữa các tập lệnh cơ bản không phải chức năng mà là kích cỡ thanh ghi cần thiết, do đó số lệnh của RV32I hay RV64I là như nhau. Đồng thời vì ta sẽ sử dụng các vi xử lý RISC-V được chia sẻ trên cộng đồng nên ta sẽ không chỉnh sửa cấu trúc của vi xử lý hay tạo ra các tập lệnh, do đó ta chỉ cần nắm được các chức năng chủ yếu của các tập lệnh mở rộng để có thể chọn một vi xử lý phù hợp làm trái tim của khối tính toán cho hệ thống trên chip của ta. Việc nghiên cứu chi tiết cấu trúc bit của từng lệnh trong các tập lệnh mở rộng là không cần thiết, bên cạnh đó thì thông số kĩ thuật của các tập lệnh này còn có thể thay đổi trong tương lai chứ không được hoàn thiện và giữ nguyên không thay đổi nữa như tập lệnh cơ bản. Tập lệnh RV32I sử dụng 32 thanh ghi, mỗi thanh ghi 32-bit, để chứa các giá trị số nguyên. Các thanh ghi này được đặt tên x0 đến x31, trong đó đặc biệt thanh ghi x0 được gán cứng giá trị 0. Tức là khi ghi dữ liệu vào x0 khi đọc ra thì giá trị vẫn là 0.

Các câu lệnh trong tập lệnh RV32I có sử dụng các thanh ghi nguồn, rs1 và rs2, và thanh ghi đích, rd, để thực hiện các phép toán. Để đơn giản hóa quá trình thiết kế phần cứng thì vị trí bit của các thanh ghi này trong các cấu lệnh là giống nhau. Trong tập lệnh cơ bản RV32I, có 6 kiểu lệnh được sắp xếp theo cấu trúc và chức năng là R, I, S, B, U và J. Cấu trúc bit của từng phiên bản được thể hiện chi tiết trong hình sau.

Format	Bit																													
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Upper immediate	imm[31:12]																				rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						

- *opcode* (7 bits): Partially specifies which of the 6 types of *instruction formats*.
- *funct7*, and *funct3* (10 bits): These two fields, further than the *opcode* field, specify the operation to be performed.
- *rs1*, *rs2*, or *rd* (5 bits): Specifies, by index, the register, resp., containing the first operand (i.e., source register), second operand, and destination register to which the computation result will be directed.

Hình 2.16 Cấu trúc bit của các kiểu lệnh trong RV32I

Từ Hình 2.16 có thể thấy, giữa các loại lệnh khác nhau, sự giống nhau của vị trí tương đối giữa các trường bit được duy trì tối đa có thể. Điều này không thay đổi chức năng của các lệnh nhưng giúp cho quá trình thiết kế vi xử lý đơn giản và dễ tối ưu hơn do các thể tận dụng một khối chức năng cho nhiều lệnh mà không cần chỉnh sửa.

Register to register là các lệnh thực thi trên các thanh ghi, thí dụ như các lệnh cộng trừ nhân, với các toán hạng là dữ liệu trên 2 thanh ghi nguồn rs1, rs2 và kết quả của phép toán sẽ được ghi vào một thanh ghi đích rd.

Immediate là các lệnh liên quan đến số, thí dụ như phép toán giữa giữ liệu trên một thanh ghi với một số. Kết quả của phép toán được ghi vào thanh ghi rd.

Store là các lệnh liên quan đến việc lưu dữ liệu vào bộ nhớ. Load là các lệnh liên quan đến việc đọc dữ liệu từ bộ nhớ. Cả hai có thể làm việc với cả bộ nhớ lệnh lẫn bộ nhớ dữ liệu.

Branch là các lệnh rẽ nhánh nhằm thực hiện các cấu trúc if else. Jump là các lệnh nhảy nhằm thực hiện các vòng lặp. Các lệnh này giúp cho việc lập trình bằng các ngôn ngữ bậc cao hơn ngôn ngữ máy trở nên dễ dàng và khả thi. Để tăng cường khoảng cách nhảy lệnh tức hỗ trợ các chương trình dài hơn thì các lệnh Upper immediate cũng được thêm vào, cho phép nhảy đến các lệnh ở khoảng cách -2^{19} đến $2^{19} - 1$ kể từ lệnh nhảy.

2.5 LiteX

LiteX là một nền tảng tạo ra bởi công ty Enjoy Digital [3] cho phép tạo ra các System on Chip và chạy trên FPGA thông qua việc sử dụng Migen, một tool cho phép thực hiện việc biên dịch python thành file verilog, cũng như cung cấp sẵn các thư viện về thư viện ngoại vi để có thể nhanh chóng tạo thành một hệ thống trên chip với lõi vi xử lý và ngoại

vi đầy đủ. LiteX cho phép gọi đến các toolchain từ thương mại như vivado hay quartus cho đến toolchain opensource như Yosys để tạo ra bitstream nạp trên FPGA. Khác với các nền tảng ChipYard không hỗ trợ nhiều cho FPGA hay nền tảng PULP được tối ưu cho việc sàn xuất ASIC, thì nền tảng LiteX được thiết kế hướng đến việc hỗ trợ mô phỏng trên FPGA nhiều nhất có thể.

LiteX cung cấp tất cả các thành phần cần thiết, thông dụng để tạo ra các hệ thống trên chip chạy trên FPGA:

- Bus và stream (Wishbone, AXI, Avalon-ST) và các interconnect
- Các lõi đơn giản: RAM, ROM, Timer, UART, JTAG, …
- Các lõi phức tạp tạo thành hệ sinh thái Litex: LiteDRAM, LitePCIe, LiteEth, LiteSATA, …
- Các lõi vi xử lý RISC-V như VexRiscV, Rocket, …
- Hỗ trợ các ngôn ngữ mô tả phần cứng VHDL, Verilog, Migen, Spinal-HDL, …
- Hỗ trợ debug thông qua Litescope
- Hỗ trợ mô phỏng qua Verilator
- Hỗ trợ backend cho toolchain open source cũng như toolchain thương mại

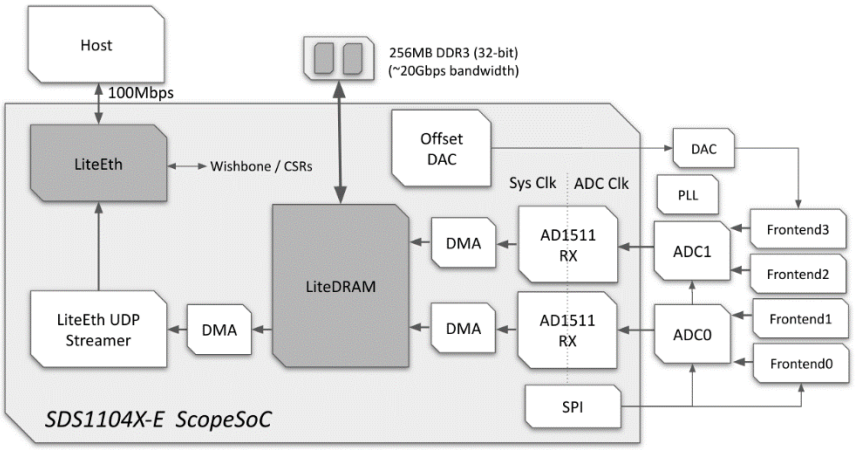
Bằng cách kết hợp LiteX với hệ sinh thái các lõi vi xử lý, việc tạo ra các SoC trở

Hình 2.17. Logo của LiteX

nên dễ dàng hơn các biện pháp truyền thống bởi vì ta chỉ cần ghép các thành phần có sẵn lại thành một hệ thống. Cách tiếp cận này cũng đem lại tính linh hoạt hơn trong việc xây dựng hệ thống trên chip khi có thể tái sử dụng nhiều thành phần một cách đơn giản để cấu hình SoC nhanh chóng mà không cần thiết phải sửa mã nguồn nhiều.

Là một nền tảng được tạo ra bởi một công ty (Enjoy Digital) và phục vụ cho các nhu cầu thương mại thì nền tảng LiteX cung cấp sự linh hoạt cao hơn so với các nền tảng trước đó. Tuy Litex chủ yếu sử dụng Migen để tạo ra các logic điện tử số nhưng đồng thời cũng cho phép tạo ra các dự án sử dụng nhiều ngôn ngữ kết hợp với nhau

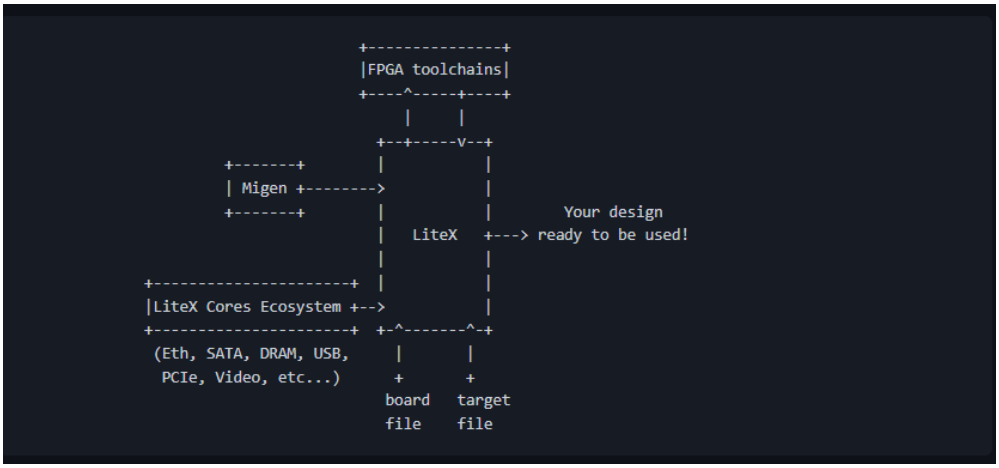
- Việc tích hợp một đoạn mã sử dụng các ngôn ngữ mô tả phần cứng khác như VHDL/Verilog/SystemVerilog/nMigen/Spinal-HDL được hỗ trợ bằng việc khởi tạo một mô đun dạng black box vào trong thiết kế và sau đó thêm các đoạn mã nguồn tương ứng vào dự án để thêm chức năng cho black box. Điều này để phục vụ cho các khách hàng chỉ muốn sử dụng nền tảng LiteX như một công cụ cho phép dễ dàng kết nối các lõi phần cứng có sẵn của họ.
- Việc sử dụng Litex để tạo ra một thiết kế, biên dịch thành Verilog và tích hợp vào trong một dự án sử dụng các ngôn ngữ mô tả phần cứng truyền thống cũng rất dễ dàng để phục vụ cho các khách hàng muốn tái sử dụng các thành phần trong thư viện phần cứng của LiteX vào thiết kế của họ.
- Hỗ trợ cho các FPGA trên một nền tảng thiết kế sử dụng chủ yếu là ngôn ngữ Python để phục vụ cho các khách hàng là những người có kiến thức với các ngôn ngữ bậc cao và không muốn phải học một ngôn ngữ mới, mà hiện tại Python là một ngôn ngữ đang rất phổ biến và ngày càng phát triển.



Hình 2.18. Hệ thống SDS1104X-E sử dụng nền tảng LiteX

2.5.1 Quy trình thiết kế của nền tảng LiteX

Để có thể sử dụng nền tảng LiteX cho việc thiết kế một hệ thống trên chip ta cần nắm được quy trình thiết kế của nền tảng đó.



Hình 2.19 Quy trình thiết kế số trên nền tảng LiteX

Như Hình 2.19 đề cập, để tạo ra một hệ thống tính toán bất kì trên nền tảng LiteX thì ta cần cung cấp cho công cụ LiteX các dữ liệu cần thiết:

- Toolchain của FPGA tương ứng, thí dụ Quartus, Vivado hay Yosys.
- Migen, là công cụ biên dịch python thành Verilog, được cung cấp sẵn trong các dự án LiteX.
- Các thư viện phần cứng LiteX Ecosystem chứa các thành phần ngoại vi có sẵn để tái sử dụng, bao gồm Ethernet, SATA, DRAM, USB, PCLE, SD Card, InterChip Communication, …
- Các board file và target file là các file cấu hình cho hệ thống mà ta cần viết sử dụng python và tuân theo quy tắc của Migen để có thể biên dịch thành công.
- Trong đó thì board file, hay còn gọi là platform file, là file chứa định nghĩa cho các thành phần của board FPGA như các IO, các ràng buộc, các xung đồng hồ, các phương thức và thành phần để nạp và chạy bitstream trên board FPGA. Còn target file là các cấu hình gốc của LiteX cho các FPGA tương ứng, với vai trò như nền tảng để tạo ra các hệ thống khác. Bằng cách khai báo thêm các thành phần cần thiết cho hệ thống của mình vào trong target file, ta có thể cấu hình nên một hệ thống mà không cần xây dựng mọi thứ từ đầu.

Sau khi cung cấp đầy đủ các dữ liệu cần thiết trên thì LiteX thực hiện quá trình biên dịch cho phép tạo ra các file Verilog và bitstream để nạp trên FPGA. Quá trình này được thực hiện một cách tự động bằng cách sử dụng ngôn ngữ Python, do đó ta chỉ cần nắm được cách thực hiện lệnh để thực hiện quá trình xây dựng hệ thống. Để có thể chỉnh sửa hệ thống như ý muốn thì ta cần nắm rõ được chức năng của các đoạn mã Python tạo nên

hệ thống, cũng như nắm rõ được các thư viện trong tuy nhiên trong phạm vi trình bày của đồ án ta sẽ không đề cập chi tiết đến việc đó.

CHƯƠNG 3. TRIỂN KHAI SNN LÊN HỆ THỐNG SOC (RISC-V) SỬ DỤNG NỀN TẢNG LITEX

3.1 Thiết kế bổ sung cho RANC để phù hợp cho việc triển khai vào hệ thống CHIP

Ở thiết kế cũ, việc nạp các thông tin (neuron parameter, neuron instruction) của mạng sẽ được cố định cứng trong code. Điều đó dẫn đến khi chúng ta ASIC hóa thì mỗi con chip chỉ có một chức năng cho việc nhận diện gì đó, không lưu động, chưa kể việc các thông số của mạng sẽ được thay đổi liên tục khi huấn luyện các đầu vào mới có độ chính xác cao hơn. Từ các lý do trên, việc dùng thiết kế cũ của mạng sẽ không ưu việt và sẽ không thể chế tạo được chip.

3.1.1 Cải thiện SRAM

Ở thiết kế cũ, SRAM sẽ được khởi tạo ban đầu với các tham số đã được nạp sẵn của quá trình huấn luyện dữ liệu. Ý tưởng sẽ là cải thiệt nó sao cho đọc ghi dữ liệu như một SRAM.

Khối neuron_grid_datapath_1x1 sẽ được thêm các tín hiệu sau:

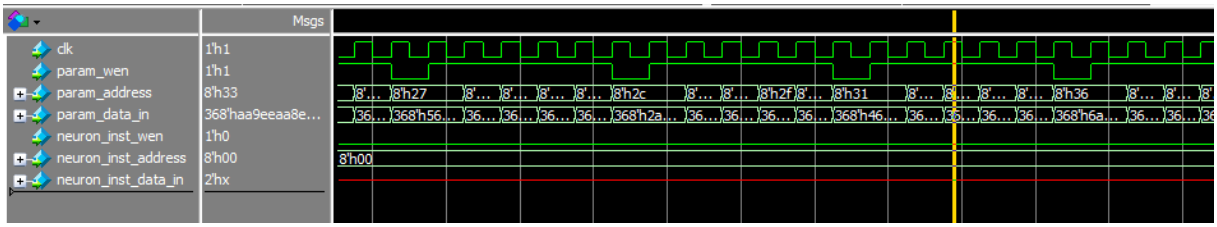
Bảng 3.1. Các tín hiệu đầu vào được thêm vào khối grid datapath

Tín hiệu	Input/Output	Kích thước	Chức năng
param_wen	Input	1	Tín hiệu cho phép ghi vào thanh ghi chứa parameter
param_address	Input	8	Địa chỉ ghi dữ liệu vào thanh ghi chứ parameter
param_data_in	Input	368	Dữ liệu ứng với địa chỉ của paramter
neuron_inst_wen	Input	1	Tín hiệu cho phép ghi vào thanh ghi chứa instruction
neuron_inst_address	Input	8	Địa chỉ ghi dữ liệu vào thanh ghi chứ instruction

neuron_inst_data_in	Input	2	Dữ liệu ứng với địa chỉ của instruction

Luồng hoạt động: Khi khởi động hệ thống, bắt buộc ban đầu sẽ phải nạp các thông số cho mạng hoạt động, ở đây sẽ là parameter và instruction. Tín hiệu wen sẽ được kích hoạt ở mức cao khi có dữ liệu muốn ghi vào, kèm với dữ liệu đó thì trên address bus phải có địa chỉ ứng với dữ liệu tương ứng. Việc ghi sẽ thực hiện trong 1 chu kỳ, khi đó tín hiệu wen sẽ được hạ xuống (Hình 3.1).

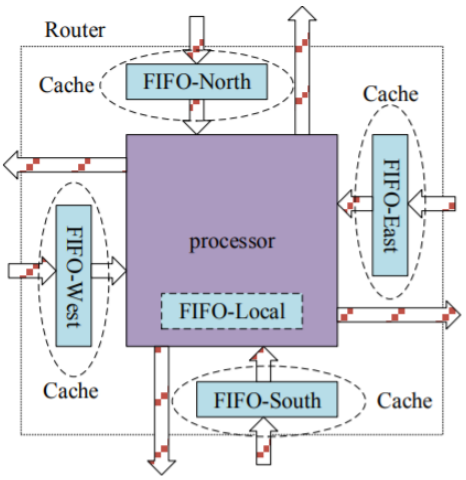
Một mạng 1 core sẽ có 256 ô nhớ cho neuron parameter và 256 ô nhớ cho neuron instruction.



Hình 3.1. Biểu đồ sóng mô tả việc ghi neuron parameter vào CSRAM

3.1.2 Tín hiệu báo hiệu sẵn sàng để gửi tick

Việc đưa ra tín hiệu báo hiệu kích hoạt tick phải được thực hiện khi tất các các packet bắn vào phải đi hết khỏi khối router (Hình 3.2) đồng nghĩa packet phải đi hết ra khỏi hướng bắc. Khi đó toàn bộ packet sẽ được chứa ở scheduler và trường **Spike Destination** sẽ bằng 0.



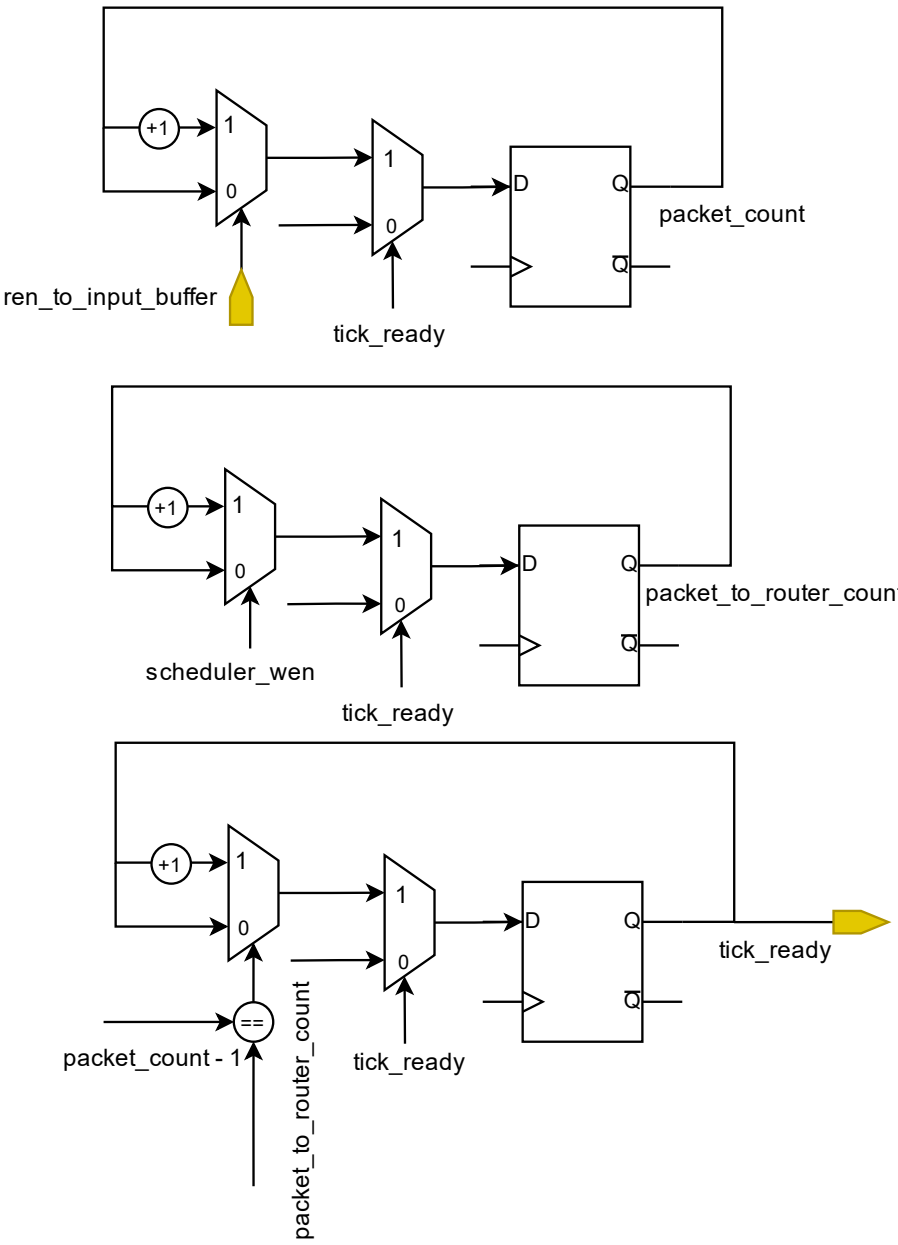
Hình 3.2. Cấu trúc của khối router

Trong khối core_1x1, tín hiệu ren_to_input_buffer là tín hiệu cho chúng ta biết có packet in bắn vào mạng. Tín hiệu scheduler_wen là tín hiệu kích hoạt cho chúng ta biết

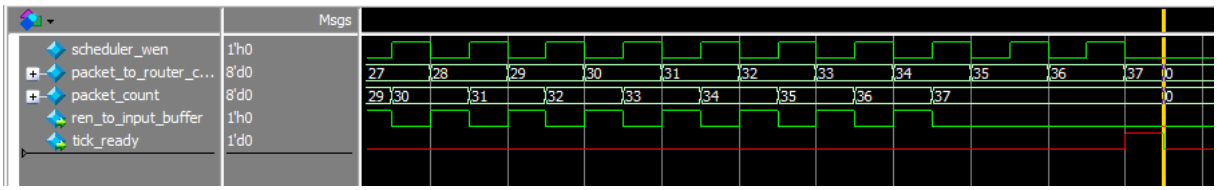
các packet được ghi vào scheduler. Nhờ có các tín hiệu này ta có thể biết được khi nào thì việc truyền các gói tin đã kết thúc và các gói tin đã đi đến Scheduler của các khối tương ứng.

Việc đưa bao nhiêu packet vào mạng sẽ được đếm bằng số lần kích hoạt của tín hiệu ren_to_input_buffer. Đồng thời số packet đã đưa qua khối scheduler sẽ được đếm bằng scheduler_wen. Tick_ready sẽ được kích hoạt khi 2 giá trị đếm trên bằng nhau (Hình 3.4).

Kiến trúc được mô tả như Hình 3.3.



Hình 3.3. Kiến trúc tạo tín hiệu tick_ready



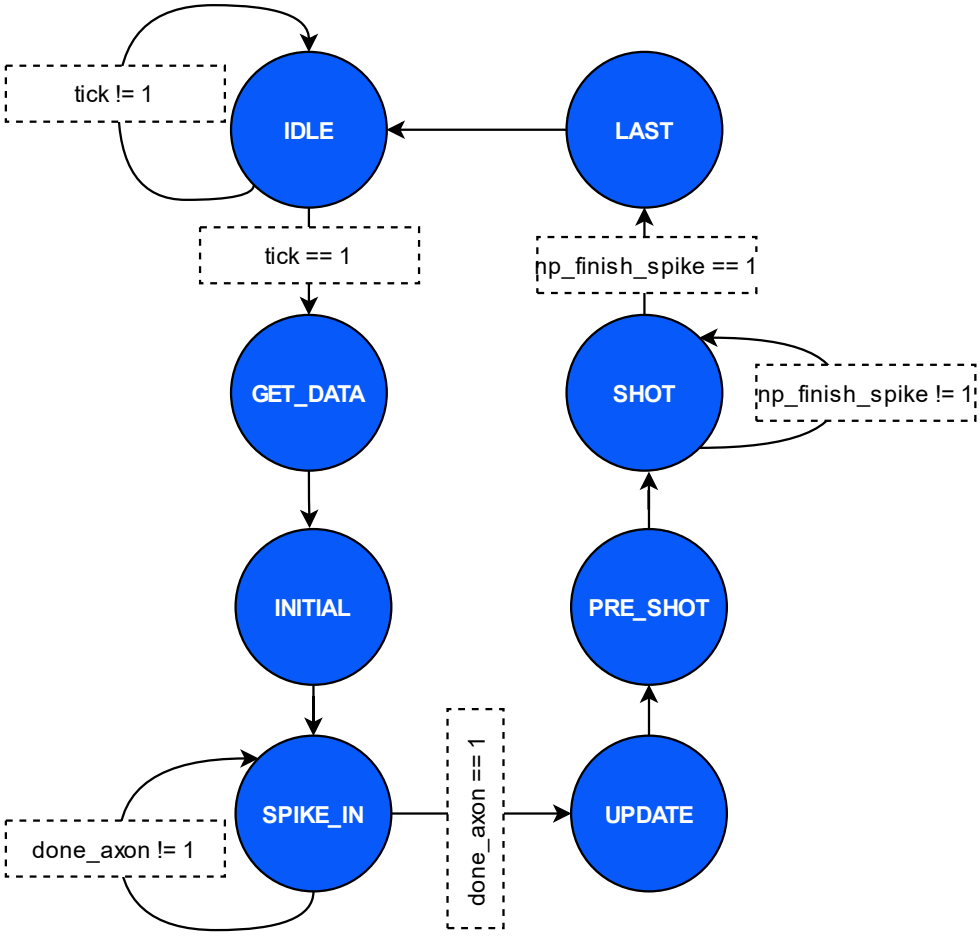
Hình 3.4. Biểu đồ sóng tạo tín hiệu tick_ready

3.1.3 Tín hiệu báo hiệu sẵn sàng để đọc packet out

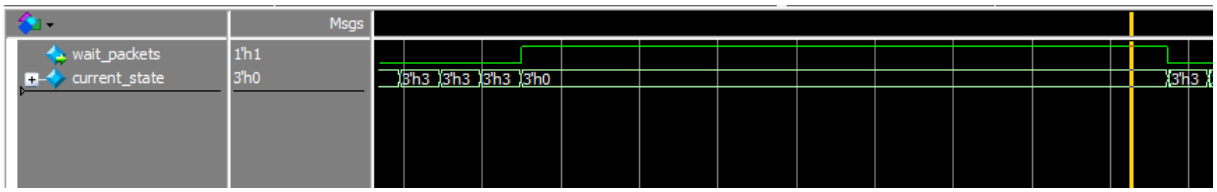
Việc báo hiệu cử lý không 1 khối packet bắn vào mạng sẽ giúp chúng ta thuận tiện cho việc đọc ghi đúng thời điểm các packet cần đọc đã được bắn ra ngoài mạng.

Hình 3.5 mô tả sơ đồ trạng thái của khối Neuron Grid Controller, cho thất mạng bắt đầu tính toán khi kích hoạt tín hiệu tick, các trạng thái sẽ được duyệt qua lần lượt để tính toán và đưa ra các packet out.

Khi tính toán xong chuỗi packet đẩy vào, máy trạng thái sẽ ở trạng thái IDLE. Dựa vào đó, đây là thời điểm đúng để đưa ra tín hiệu đọc packet out (wait_packets).



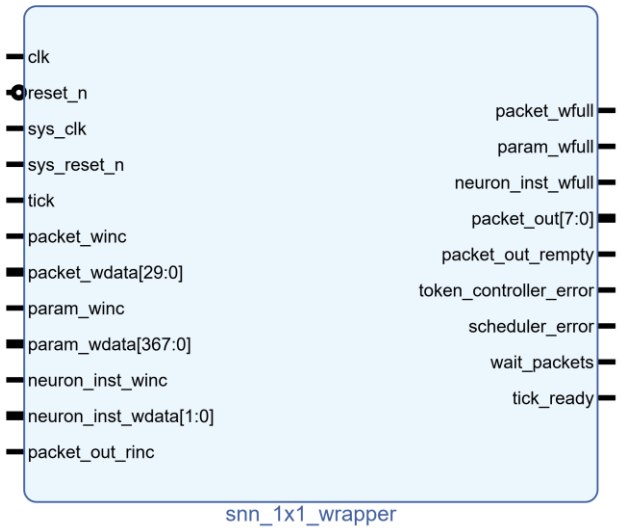
Hình 3.5. Sơ đồ máy trạng thái của Neuron Grid Controller



Hình 3.6. Biểu đồ sóng tạo tín hiệu sẵn sàng để đọc packet out

3.2 Thiết kế khối SNN wrapper

SNN wrapper là khối chính của hệ thống chứa mạng SNN. Các tín hiệu điều khiển vào ra được mô tả ở Bảng 3.2.



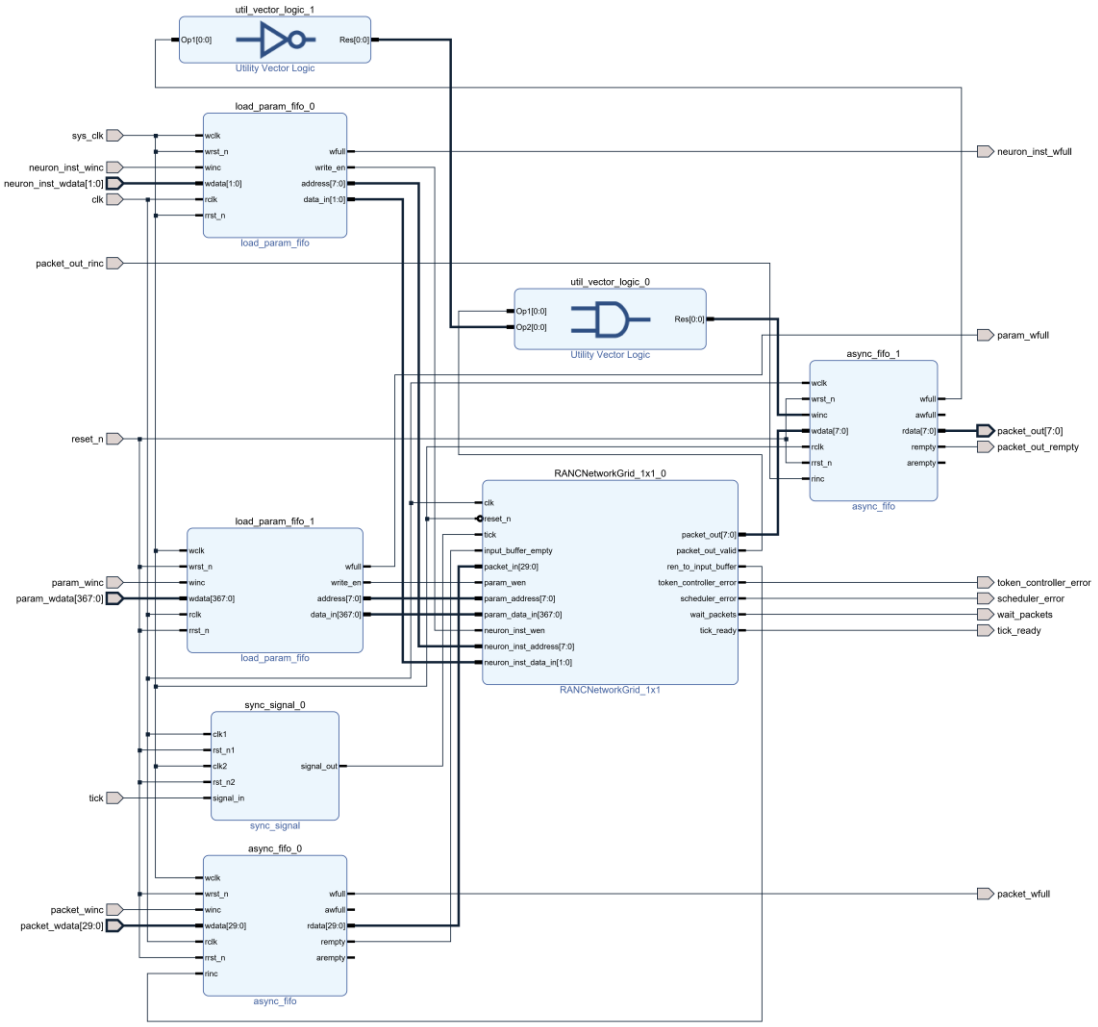
Hình 3.7. Sơ đồ khối của khối SNN wrapper

Bảng 3.2. Chức năng của các tín hiệu vào ra trong khối SNN wrapper

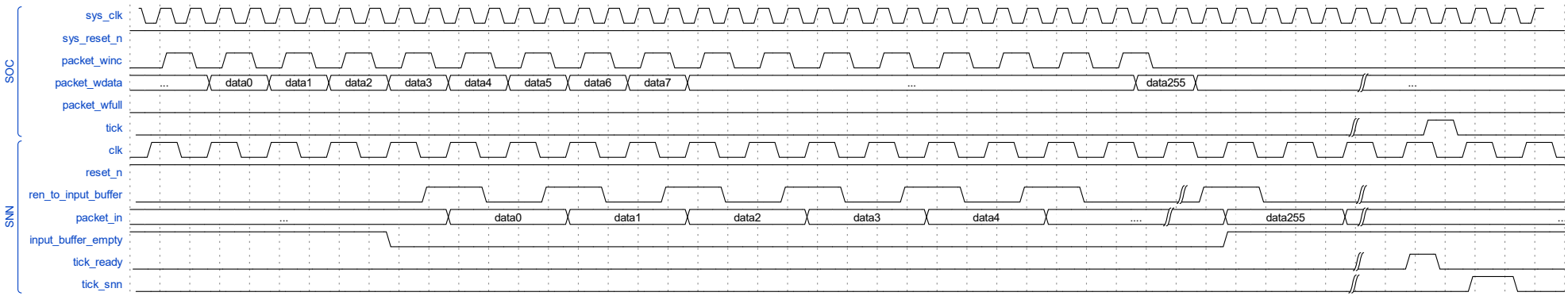
Tín hiệu	Input/Output	Độ rộng	Chức năng
clk	Input	1	Tín hiệu đồng hồ cung cấp cho SNN
reset_n	Input	1	Tín hiệu reset không đồng bộ cho SNN
sys_clk	Input	1	Tín hiệu đồng hồ cấp cho hệ thống SOC
sys_reset_n	Input	1	Tín hiệu reset không đồng bộ cho SOC
tick	Input	1	Tín hiệu kích hoạt
packet_winc	Input	1	Kích hoạt ghi packet vào hệ thống

packet_wdata	Input	30	Spike packet đầu vào
packet_wfull	Output	1	Cờ chỉ ra packet FIFO bị đầy
param_winc	Input	1	Kích hoạt ghi parameter
param_wdata	Input	368	Dữ liệu Neuron parameter
param_wfull	Output	1	Cờ chỉ ra parameter FIFO bị đầy
neuron_inst_winc	Input	1	Kích hoạt ghi Neuron instruction
neuron_inst_wdata	Input	2	Dữ liệu Neuron instruction
neuron_inst_wfull	Output	1	Cờ chỉ ra neuron instruction FIFO bị đầy
packet_out	Output	8	Spike packet đầu ra
packet_out_rinc	Input	1	Kích hoạt đọc packet đầu ra
packet_out_rempty	Output	1	Cờ chỉ ra packet out FIFO rỗng
token_controller_error	Output	1	Cờ chỉ ra token controller bị lỗi
scheduler_error	Output	1	Cờ chỉ ra scheduler bị lỗi
wait_packets	Output	1	Tín hiệu chỉ ra đang đợi packet đẩy vào mạng hoặc là khoảng thời gian đọc packet đầu ra đúng
tick_ready	Output	1	Cờ chỉ ra có thể đẩy tín hiệu kích hoạt để hệ thống bắt đầu tính toán

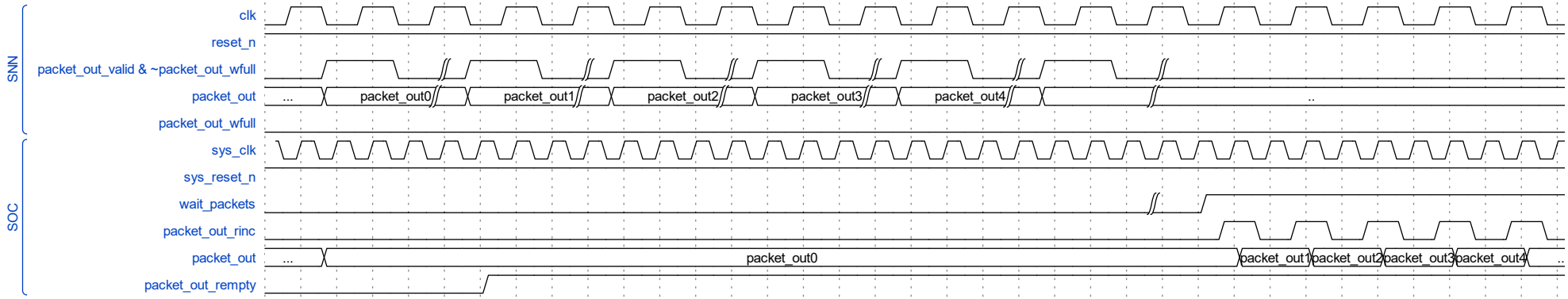
Do tần số của hệ thống thường sẽ cao hơn rất nhiều so với SNN nên để đồng bộ 2 miền tần số, các asynchronous FIFO [4] được dùng để nạp và đọc dữ liệu từ khối SNN. Kiến trúc được mô tả như Hình 3.8.



Hình 3.8. Kiến trúc tổng quát của SNN wrapper



Hình 3.9. Biểu đồ sóng quá trình nạp packet vào mạng



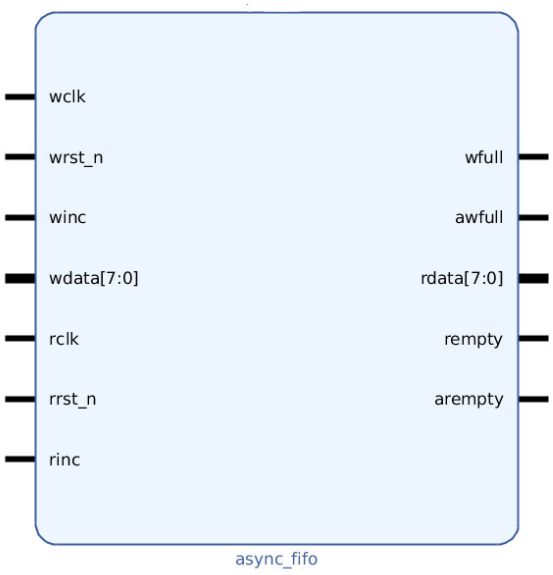
Hình 3.10. Biểu đồ sóng quá trình đọc packet từ mạng

- Quá trình ghi packet: Khi có gói dữ liệu muốn đẩy vào mạng, tín winc sẽ được kích hoạt để ghi vào FIFO. Lúc đó tín hiệu input_buffer_empty sẽ được kích hoạt (trạng thái empty của FIFO), mạng sẽ trả về tín hiệu ren_to_input_buffer để đọc dữ liệu packet vào mạng. Khi ghi đủ số packet, sau 1 khoảng thời gian tín hiệu tick_ready sẽ được kích hoạt, lúc đó chính là thời điểm chúng ta kích hoạt tính toán (tick).
- Quá trình đọc packet: Trong quá trình tính toán, các packet ra hợp lệ sẽ đi cùng packet_out_valid. Lúc đó packet đó sẽ được ghi vào FIFO trừ những lúc FIFO đã đầy. Hệ thống sẽ chờ đến khi nào tín hiệu wait_packets được kích hoạt, lúc đó sẽ điều khiển packet_out_inc để đọc hết packet lưu trong FIFO.

Quá trình đọc packet và ghi packet được mô tả như Hình 3.9 và Hình 3.10.

3.2.1 Asynchronous fifo

FIFO này sẽ thuận tiện trao đổi dữ liệu giữa 2 miền tần số khác nhau. Khối sẽ tự động quản lý địa chỉ RAM nội bộ và thông báo cho người dùng về trạng thái của nó, ở đây là full, empty,...



Hình 3.11. Sơ đồ khối tổng quát của khối Asynchronous fifo

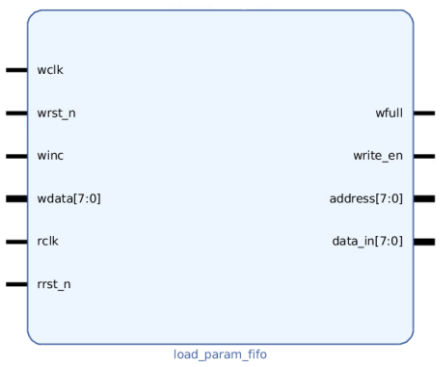
Các chức năng của chân vào ra được thể hiện như Bảng 3.4.

Bảng 3.3. Tín hiệu vào ra của khối Asynchronous fifo

Tín hiệu	Input/Output	Độ rộng	Chức năng
wclk	Input	1	Tín hiệu đồng hồ bên ghi
wrst_n	Input	1	Tín hiệu reset không đồng bộ cho bên ghi
winc	Input	1	Tín hiệu cho phép ghi vào FIFO
wdata	Input	DSIZE	Dữ liệu ghi vào FIFO
wfull	Output	1	Tín hiệu báo FIFO đã đầy
rclk	Input	1	Tín hiệu đồng hồ bên đọc
rrst_n	Input	1	Tín hiệu reset không đồng bộ cho bên đọc
rinc	Input	1	Tín hiệu cho phép đọc dữ liệu
rdata	Output	DSIZE	Dữ liệu đầu ra
rempty	Output	1	Tín hiệu báo FIFO đang rỗng

3.2.2 Khối load_param_fifo

Khối load_param_fifo dùng để nạp các parameter và instruction vào mạng. Khối sẽ tự động tạo ra địa chỉ sao cho phù hợp với thứ tự các dữ liệu nạp vào CSRAM. Cấu trúc được mô tả như Hình 3.12.



Hình 3.12. Sơ đồ khối của khối load_param_fifo

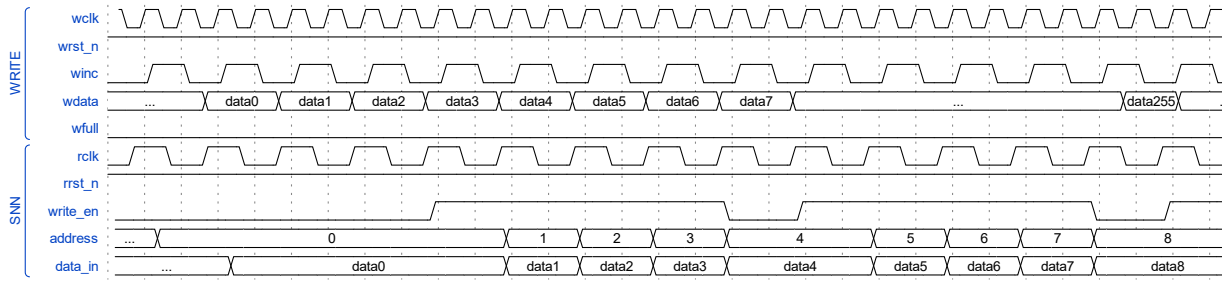
Chức năng các tín hiệu vào ra được mô tả chi tiết ở Bảng 3.4.

Bảng 3.4. Tín hiệu vào ra của khối load_param_fifo

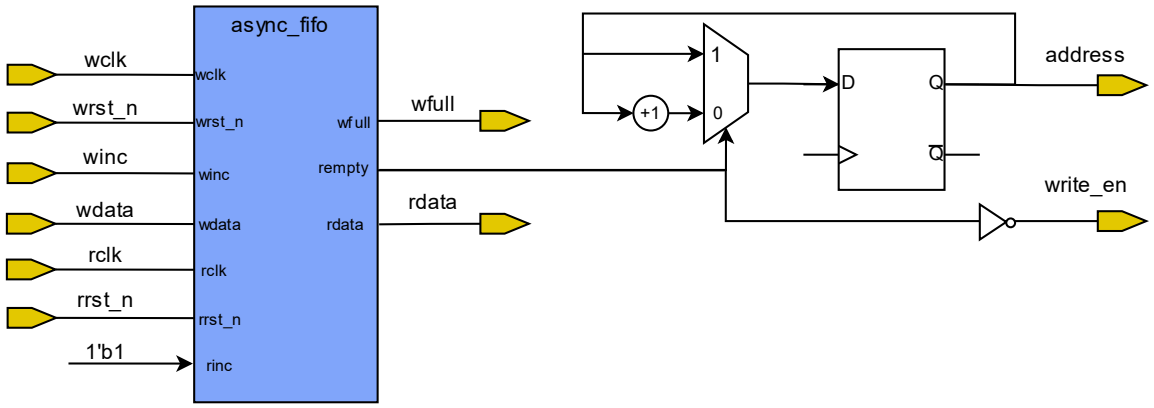
Tín hiệu	Input/Output	Độ rộng	Chức năng
wclk	Input	1	Tín hiệu đồng hồ bên ghi
wrst_n	Input	1	Tín hiệu reset không đồng bộ cho bên ghi
winc	Input	1	Tín hiệu cho phép ghi vào FIFO
wdata	Input	DSIZE	Dữ liệu ghi vào FIFO
wfull	Output	1	Tín hiệu báo FIFO đã đầy
rclk	Input	1	Tín hiệu đồng hồ bên đọc
rrst_n	Input	1	Tín hiệu reset không đồng bộ cho bên đọc
write_en	Output	1	Tín hiệu cho phép ghi vào CSRAM
address	Output	8	Địa chỉ của dữ liệu muốn ghi vào CSRAM
data_in	Output	DSIZE	Dữ liệu muốn ghi vào CSRAM ứng với địa chỉ address

Dữ liệu được ghi vào FIFO được kích hoạt bằng tín hiệu winc. Do có sự khác nhau giữa 2 tần số, tín hiệu empty sẽ được hạ xuống sau 3 chu kỳ ở bên có tần số thấp. Tín hiệu write_en thực chất sẽ là đảo của tín hiệu empty, nó sẽ cho phép các dữ liệu tồn tạo trong FIFO vào CSRAM, đồng thời nó cũng sẽ là tín hiệu đọc phần tử của FIFO. Các địa chỉ tương ứng sẽ được sinh ra bằng 1 bộ đếm. Mô tả các hoạt động như Hình 3.13.

Kiến trúc có sử dụng asynchronous FIFO để làm nơi trung chuyển dữ liệu, kiến trúc được mô tả như hình Hình 3.14.



Hình 3.13. Biểu đồ sóng mô tả quá trình ghi parameter và instruction vào mạng



Hình 3.14. Kiến trúc của khối load_param_fifo

3.3 Các thanh ghi CSR để điều khiển hoạt động khối mạng Neuron

Để khối tính toán là các vi xử lý Vexriscv có thể điều khiển hoạt động của mạng Neuron, ta cần một giao thức cho phép truyền lệnh và gửi dữ liệu kết quả trả về giữa các khối này. Giải pháp được chọn ở đâu là tạo ra các thanh ghi trạng thái và dữ liệu với sự hỗ trợ của LiteX. Sử dụng kết nối nội bộ sẽ là wishbone.

Bảng 3.5. Mô tả các thanh ghi có trong hệ thống

Thanh ghi	Độ rộng	Chức năng
EDABK_SNN_TICK	1	Kích hoạt tính toán cho mạng
EDABK_SNN_PACKET_WDATA	20	Dữ liệu của packet bắn vào mạng

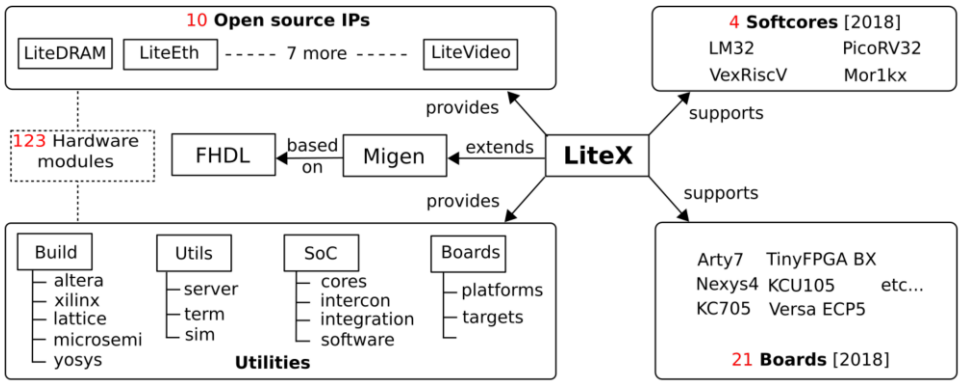
Thanh ghi	Độ rộng	Chức năng
EDABK_SNN_PARAM_WDATA0	32	Dữ liệu của parameter đưa vào mạng
EDABK_SNN_PARAM_WDATA1	32	
EDABK_SNN_PARAM_WDATA2	32	
EDABK_SNN_PARAM_WDATA3	32	
EDABK_SNN_PARAM_WDATA4	32	
EDABK_SNN_PARAM_WDATA5	32	
EDABK_SNN_PARAM_WDATA6	32	
EDABK_SNN_PARAM_WDATA7	32	
EDABK_SNN_PARAM_WDATA8	32	
EDABK_SNN_PARAM_WDATA9	32	
EDABK_SNN_PARAM_WDATA10	32	
EDABK_SNN_PARAM_WDATA11	16	
EDABK_SNN_NEURON_INST_WDATA	2	Dữ liệu của instruction nạp vào mạng
EDABK_SNN_PACKET_OUT_RINC	1	Kích hoạt đọc dữ liệu packet đầu ra

Thanh ghi		Độ rộng	Chức năng
EDABK_SNN_PACKET_OUT		8	Dữ liệu của packet đầu ra
EDABK_SNN_TICK_READY		1	Cho biết đã sẵn sàng kích hoạt quá trình tính toán
EDABK_SNN_SNN_STATUS	PACKET_WFULL	1	FIFO chứa packet đầu vào bị đầy
	PARAM_WFUL	1	FIFO chứa parameter bị đầy
	NEURON_INST_WFULL	1	FIFO chứa instruction bị đầy
	TOKEN_CONTROLLER_ERROR	1	Token controller bị lỗi
	WAIT_PACKETS	1	Khoảng thời gian đọc packet đầu ra đúng
	SCHEDULER_ERROR	1	Scheduler bị lỗi

Bảng 3.5 liệt kê các thanh ghi sử dụng trong hệ thống. Ở tín hiệu PARAM_DATA sẽ được chia thành 12 thanh ghi do độ rộng tối đa của CSR là 32 bit.

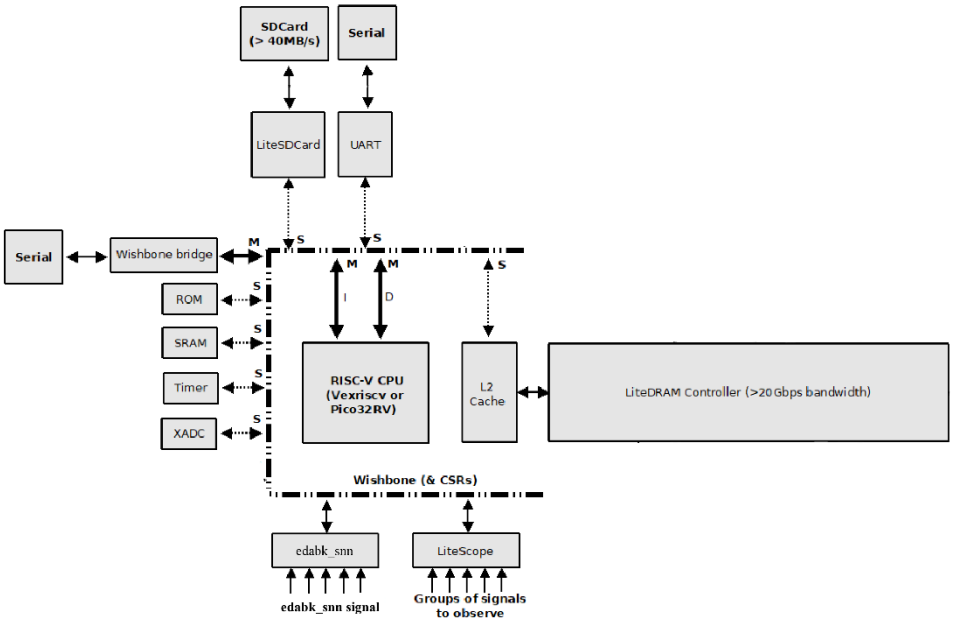
3.4 Tích hợp mạng Neuron vào hệ thống trên chip

Để có thể tích hợp mạng Neuron trên vào trong hệ thống trên, ta sẽ cần sử dụng sự hỗ trợ của nền tảng LiteX. Nền tảng này cho phép tích hợp hay tái sử dụng các khối chức năng hay các lõi vi xử lý (mã nguồn mở và cả thương mại) vào trong các thiết kế có sẵn của nền tảng, giúp dễ dàng tạo ra các thiết kế đa ngôn ngữ.



Hình 3.15. Tổng quan về LiteX

Hình 3.16 mô tả về kiến trúc của hệ thống đang thiết kế. Việc kết nối SNN với hệ thống sẽ được trung gian qua registerblocks, giao tiếp đọc ghi dữ liệu sẽ thông qua chuẩn wishbone.



Hình 3.16. Kiến trúc hệ thống

Do chưa triển khai được camera lên hệ thống để lấy ảnh thời gian thực, chúng ta sẽ sử dụng SDcard để lưu trữ dữ liệu dưới đạng text, sau đó thực hiện đọc file để đưa các

thông số (parameter, instruction, packet in) đẩy vào mạng neuron. Giao tiếp đó sẽ được vi xử lý thực hiện.

Các thanh ghi cho bộ SNN sẽ có các địa chỉ riêng biệt (Bảng 3.6). Muốn đọc hoặc ghi dữ liệu các thanh ghi thì cần truyền đúng địa chỉ tương ứng với các thanh ghi.

Bảng 3.6. Địa chỉ ứng với các thanh ghi của mạng neuron

Register	Address
EDABK_SNN_TICK	0xf0000000
EDABK_SNN_PACKET_WDATA	0xf0000004
EDABK_SNN_PARAM_WDATA0	0xf0000008
EDABK_SNN_PARAM_WDATA1	0xf000000c
EDABK_SNN_PARAM_WDATA2	0xf0000010
EDABK_SNN_PARAM_WDATA3	0xf0000014
EDABK_SNN_PARAM_WDATA4	0xf0000018
EDABK_SNN_PARAM_WDATA5	0xf000001c
EDABK_SNN_PARAM_WDATA6	0xf0000020
EDABK_SNN_PARAM_WDATA7	0xf0000024
EDABK_SNN_PARAM_WDATA8	0xf0000028
EDABK_SNN_PARAM_WDATA9	0xf000002c
EDABK_SNN_PARAM_WDATA10	0xf0000030
EDABK_SNN_PARAM_WDATA11	0xf0000034
EDABK_SNN_NEURON_INST_WDATA	0xf0000038
EDABK_SNN_PACKET_OUT_RINC	0xf000003c
EDABK_SNN_PACKET_OUT	0xf0000040
EDABK_SNN_TICK_READY	0xf0000044
EDABK_SNN_SNN_STATUS	0xf0000048

3.4.1 Sử dụng migen để tạo khối SNN và CSR

Quá trình khởi tạo khối chức năng đơn giản là dựa vào cú pháp khai báo các khối logic của Migen.

Khởi tạo các thanh ghi như Hình 3.17, kết nối giữa thanh ghi và khối snn_wrapper được mô tả như Hình 3.18.

```
self.clk          = Signal()
self.sys_clk      = Signal()
self.param_wdata  = Signal(384)
self.tick         = CSRStorage(name="tick",description='Send tick to SNN', reset=0x0, size=1)
self.packet       = CSRStorage(name="packet_wdata",description='Packet data send SNN', reset=0x0, size=30)

self.param0       = CSRStorage(name="param_wdata0",description='Param data0 send SNN', reset=0x0, size=32)
self.param1       = CSRStorage(name="param_wdata1",description='Param data1 send SNN', reset=0x0, size=32)
self.param2       = CSRStorage(name="param_wdata2",description='Param data2 send SNN', reset=0x0, size=32)
self.param3       = CSRStorage(name="param_wdata3",description='Param data3 send SNN', reset=0x0, size=32)
self.param4       = CSRStorage(name="param_wdata4",description='Param data4 send SNN', reset=0x0, size=32)
self.param5       = CSRStorage(name="param_wdata5",description='Param data5 send SNN', reset=0x0, size=32)
self.param6       = CSRStorage(name="param_wdata6",description='Param data6 send SNN', reset=0x0, size=32)
self.param7       = CSRStorage(name="param_wdata7",description='Param data7 send SNN', reset=0x0, size=32)
self.param8       = CSRStorage(name="param_wdata8",description='Param data8 send SNN', reset=0x0, size=32)
self.param9       = CSRStorage(name="param_wdata9",description='Param data9 send SNN', reset=0x0, size=32)
self.param10      = CSRStorage(name="param_wdata10",description='Param data10 send SNN', reset=0x0, size=32)
self.param11      = CSRStorage(name="param_wdata11",description='Param data11 send SNN', reset=0x0, size=16)

self.neuron_inst     = CSRStorage(name="neuron_inst_wdata",description='neuron_inst data send SNN', reset=0x0, size=2)

self.packet_out_rinc = CSRStorage(name="packet_out_rinc",description='Enable signal read packet out data', reset=0x0, size=1)
self.packet_out      = CSRStorage(name="packet_out",description='Packet out data', reset=0x0, size=8, write_from_dev=True)
self.tick_ready      = CSRStorage(name="tick_ready", description="Tick ready", size=1, reset=0x0, write_from_dev=True)

self.snn_status = CSRStatus(size=32, fields=[
    CSRField(name="packet_wfull", description="flag full", size=1, reset=0x0),
    CSRField(name="param_wfull", description="flag full", size=1, reset=0x0),
    CSRField(name="neuron_inst_wfull", description="flag full", size=1, reset=0x0),
    CSRField(name="packet_out_rempty", description="Packet out data is empty", size=1, reset=0x0),
    CSRField(name="token_controller_error", description="Token controller error", size=1, reset=0x0),
    CSRField(name="scheduler_error", description="Scheduler error", size=1, reset=0x0),
    CSRField(name="wait_packets", description="Wait for packet to put on snn/ Time to read packetout", size=1, reset=0x0)
], description="SNN status")
```

Hình 3.17. Code mô tả tạo thanh ghi

```
    self.comb += self.param_wdata.eq(Cat(self.param0.storage,
        self.param1.storage ,
        self.param2.storage ,
        self.param3.storage ,
        self.param4.storage ,
        self.param5.storage ,
        self.param6.storage ,
        self.param7.storage ,
        self.param8.storage ,
        self.param9.storage ,
        self.param10.storage,
        self.param11.storage
    ))

    self.comb += self.packet_out.we.eq(1)
    self.comb += self.tick_ready.dat_w.eq(1)

    self.specials += Instance(
        "snn_1x1_wrapper"                                                    ,
        i_clk                    = self.clk                                  ,
        i_reset_n                = ~ResetSignal()                            ,
        i_sys_clk                = self.sys_clk                              ,
        i_sys_reset_n            = ~ResetSignal()                            ,
        i_tick                   = self.tick.re                              ,
        i_packet_winc            = self.packet.re                            ,
        i_packet_wdata           = self.packet.storage                       ,
        o_packet_wfull           = self.snn_status.fields.packet_wfull       ,
        i_param_winc             = self.param11.re                           ,
        i_param_wdata            = self.param_wdata                          ,
        o_param_wfull            = self.snn_status.fields.param_wfull        ,
        i_neuron_inst_winc       = self.neuron_inst.re                       ,
        i_neuron_inst_wdata      = self.neuron_inst.storage                  ,
        o_neuron_inst_wfull      = self.snn_status.fields.neuron_inst_wfull  ,
        o_packet_out             = self.packet_out.dat_w                     ,
        i_packet_out_rinc        = self.packet_out_rinc.re                   ,
        o_packet_out_rempty      = self.snn_status.fields.packet_out_rempty  ,
        o_token_controller_error = self.snn_status.fields.token_controller_error,
        o_scheduler_error        = self.snn_status.fields.scheduler_error    ,
        o_wait_packets           = self.snn_status.fields.wait_packets       ,
        o_tick_ready             = self.tick_ready.we
    )
```

Hình 3.18. Code mô tả kết nối giữa module SNN_wrapper với thanh ghi

3.4.2 Triển khai hệ thống bằng migen

Hệ thống sử dụng:

- Kiến trúc RISC-V: Vexriscv
- LiteX SDCARD
- RAM DDR3
- Tần số hệ thống: 125Mhz
- Tần số cho SNN: 100Mhz

Code mô tả bằng ngôn ngữ migen như Hình 3.19 ứng với kiến trúc (Hình 3.16).

```
class BaseSoC(SoCCore):
    def __init__(self, sys_clk_freq=int(125e6), with_ethernet=False, with_sdcard=False, with_led_chaser=True,
                 with_spi_flash=False, with_pcie=False, with_sata=False, **kwargs):
        platform = xilinx_kc705.Platform()

        # CRG --------------------------------------------------------------------------------------
        self.submodules.crg = _CRG(platform, sys_clk_freq)

        # SoCCore ----------------------------------------------------------------------------------
        SoCCore.__init__(self, platform, sys_clk_freq, ident="LiteX SoC on KC705", **kwargs)

        # SDCard -----------------------------------------------------------------------------------
        # Simply integrate SDCard through LiteX's add_sdcard method.
        if with_sdcard:
            self.add_sdcard()

        # EDABK SNN--------------------------------------------------------------------------------
        # SNN clk, Generate 100MHz from PLL.
        self.clock_domains.cd_snn_clk = ClockDomain()
        self.crg.pll.create_clkout(self.cd_snn_clk, 100e6)
        snn_clk = ClockSignal("snn_clk")
        # platform.add_platform_command("set_property SEVERITY {{Warning}} [get_drc_checks REQP-52]")

        self.submodules.edabk_snn = edabk_snn(self.platform)
        self.add_csr("edabk_snn")
        self.comb += self.edabk_snn.clk.eq(snn_clk)
        self.comb += self.edabk_snn.sys_clk.eq(ClockSignal())

        # DDR3 SDRAM -------------------------------------------------------------------------------
```

Hình 3.19. Code mô tả hệ thống

3.5 Triển khai trên kit FPGA

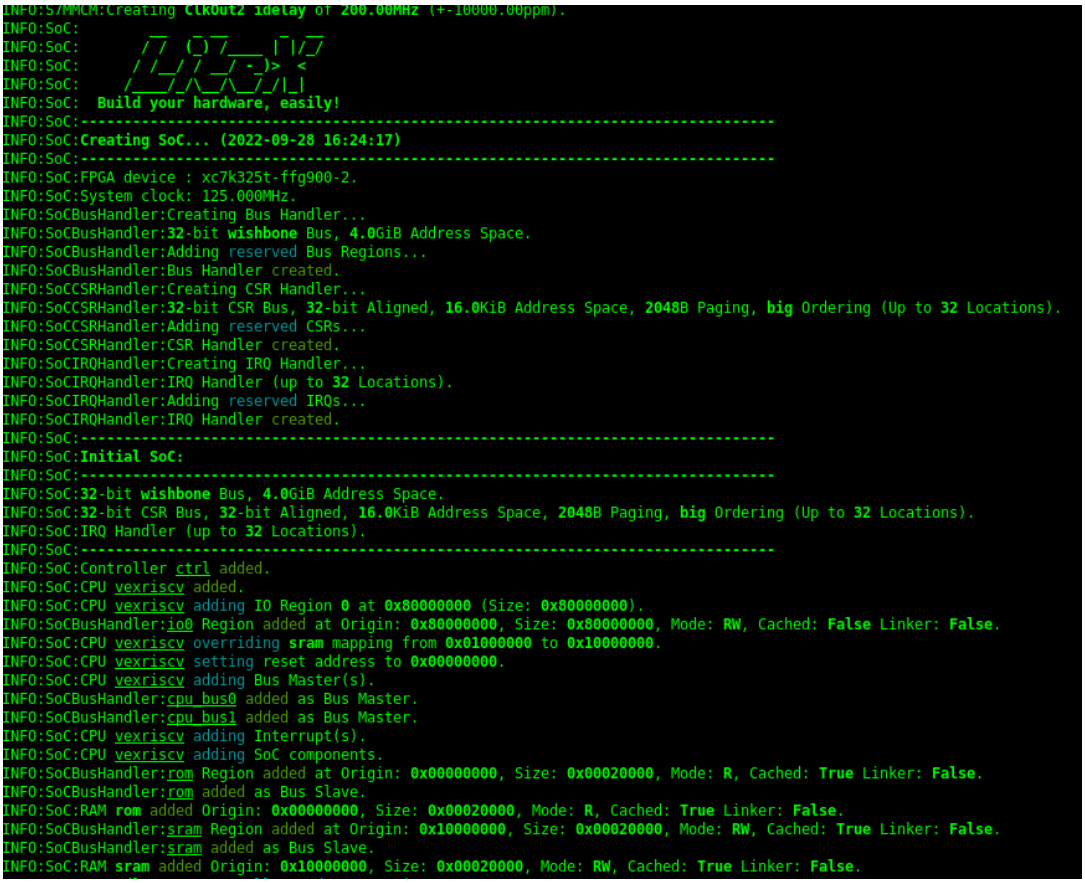
Hệ thống sẽ được thực hiện trên kit: Xilinx KC705

Đầu tiên sẽ tạo hệ thống:

```
python3 soc.py --build --with-sdcard --integrated-sram-size=0x20000 --doc
```

Kết quả như Hình 3.20. Lúc đó các file bitstream sẽ được tạo ra và sẽ được nạp vào kit thông qua lệnh:

```
python3 soc.py --load
```



Hình 3.20. Màn hình hiển thị quá trình tạo

Việc thực hiện đọc ghi từ thẻ SD và chuyển vào mạng sẽ thông qua Bare Metal. Fireware sẽ được nạp thông qua giao thức UART.

Thực hiện compile các tệp được viết bằng ngôn ngữ C.

```
python3 firmware.py --build-path='../build/xilinx_kc705
```

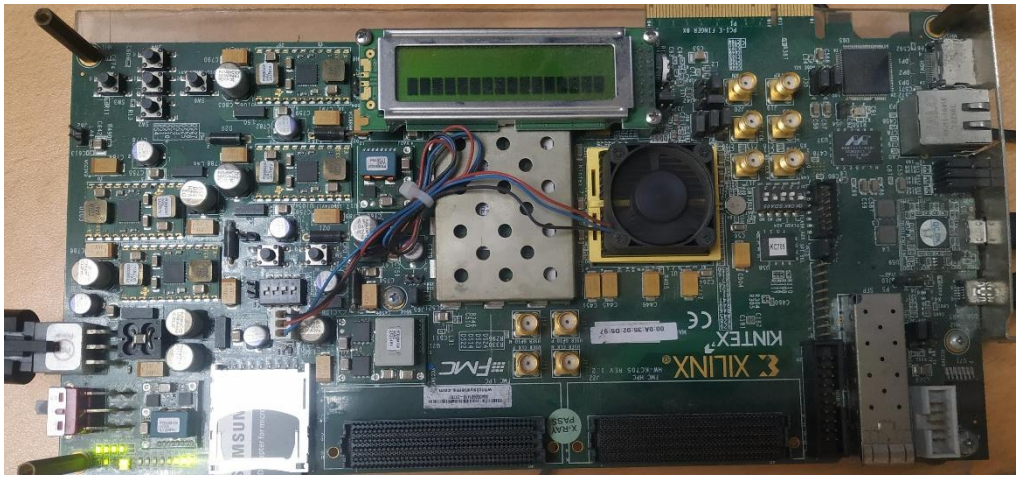
Kết quả sau quá trình đó sẽ là firmware.bin, tiến hành nạp vào hệ thống:

```
[> Build and Load over LiteX-Term

---------------------------------

litex_term /dev/ttyUSB0 --kernel=firmware.bin
```

Giao diện khởi động như Hình 3.21. Để chạy các chức năng hỗ trợ thì phải nhập đúng lệnh được hỗ trợ được hiển thị trong mục help.

```
Switching SDRAM to hardware control.
Memtest at 0x40000000 (2.0MiB)...
  Write: 0x40000000-0x40200000 2.0MiB
   Read: 0x40000000-0x40200000 2.0MiB
Memtest OK
Memspeed at 0x40000000 (Sequential, 2.0MiB)...
  Write speed: 112.6MiB/s
   Read speed: 93.6MiB/s

--============== Boot ==================--
Booting from serial...
Press Q or ESC to abort boot completely.
sL5DdSMmkekro
[LITEX-TERM] Received firmware download request from the device.
[LITEX-TERM] Uploading firmware.bin to 0x40000000 (19220 bytes)...
[LITEX-TERM] Upload calibration... (inter-frame: 10.00us, length: 64)
[LITEX-TERM] Upload complete (9.9KB/s).
[LITEX-TERM] Booting the device.
[LITEX-TERM] Done.
Executing booted program at 0x40000000

--============== Liftoff! ===============--
IniInitializing SDCard...
SDCard init successfully!

LiteX minimal demo app built Sep 28 2022 16:32:21

Available commands:
help               - Show this command
reboot             - Reboot CPU
test_snn           - Test SNN function
show_sd_info       - Show SDCard info
led                - Led demo
donut              - Spinning Donut demo
helloc             - Hello C
edabk-snn-app>
```

Hình 3.21. Giao diện khởi động của hệ thống



Hình 3.22. Kết quả trên kit trực tiếp

3.6 Kiểm thử hệ thống

Thực hiện đọc 10 file đầu vào chứa các packet đầu vào, sau đó so sánh các packet được bắn ra với kết quả của quá trình mô phỏng.

```
edabk-snn-app> test_snn
Parameter   0:     EAAF EAAA292B    2B34DA 925752A6 D292DA86 9EC79443 8A23BA50 2B6AAAAC 74670000   3FF00F FFFE0000     1000
Parameter   1:    9ABA1 2AA960A3 AAAAD05B D14BA94B 890A990E A12EA9AF AAAE8AAB 130A2029 A5B70000   3FF00F FC000000     1010
Parameter   2:    EAAC1 24A955A7 540CD65D 52585056 D0325893 DA96BAD7 A356A954 2152E952 658B0000   3FF00F FC000000     1020
Parameter   3:     AAE1 6ED23DF5 9D62116C A42D86AD A32FA5AA 2DE8AADB AAD6A952 BD53A8EE 4E990000   3FF00F FC020000     1030
Parameter   4:    2AA21 E8A840AA 50A23490 8497A0CB A6C78683 A6A2B687 B6B39286 53092AA8 66950000   3FF00F FFFC0000     1040
Parameter   5:    6AA91 1245D403 520350AC C2AA82AB 10AAA277 2ADE6AC3 E84A551B D529B0AF BFEE0000   3FF00F FFFC0000     1050
Parameter   6:    AA8D1 2AA9A4A9 904A904A C04AF65B 325BBE4B AA6BAA2B 60AA82A8 10A1D777 3A6C0000   3FF00F FFFE0000     1060
Parameter   7:    5C461 2B24AA76 EA632A0B AA4B2143 712BA8A8 28A92698 3451AC50 2AA96AAA 720B0000   3FF00F FFFC0000     1070
Parameter   8:    4AAC1 E22AA92F 28EE206B A46BA82B AB2BA90E 2446A6DB 2683CA1B 682B2EAF F1C20000   3FF00F FFFA0000     1080
Parameter   9:     55D1 91549A53 8BBA4AA2 AB4AB74A B56F95A7 B4AEA883 A54AA0A8 2AAB2AA9 26FA0000   3FF00F FFFE0000     1090
Parameter  10:    CAA31 2AAA2B4B A14AA8CB 86929283 9A82C292 828A9232  48AA128 AC2AAAAE 945E0000   3FF00F FFFE0000     10A0
Parameter  11:    AAAB1 6AAB7502 F517546B D24BAA0B  92B2DB7 A5ABA8AF A27EA953 215AAA23 44EA0000   3FF00F FC020000     10B0
Parameter  12:    EAAF1 A5ABD4B6 D4BE568A 504B5157 D89F5EBA B296B696 BB52A952 AB50A942 57BA0000   3FF00F FC020000     10C0
Parameter  13:    EAAB1 6A5629D4 22A424AD B0ADBA48 A94B2AAA A2ACB414 21562057 2AB7AAAA 56650000   3FF00F FFFE0000     10D0
Parameter  14:    8AAA1 824A304B 80AB28AF A957A917 2917A00B B04A9423 94A29483 D28B2AA8 DAE80000   3FF00F FC000000     10E0
Parameter  15:    EAAA1 EAC8A9CE AD472C11 A4AEB8AB ABAEAA4B CB4ADB0B C3AB122B 5E69AEA9 8E5F0000   3FF00F FC020000     10F0
Parameter  16:    AAA21 6AABA829 A84AA78A A47BB07B 3CFB9A7B AA3AA9BA 28BAB4BA 968A17D0 2A3F0000   3FF00F FFFE0000     1100
Parameter  17:    90001 6E17A956 AA57282B 8CAB86AF A4A9A1A8 A56D29D5 B151A851 AAA8AAAB FF900000   3FF00F FC020000     1110
Parameter  18:    2AAC1 EBAB3D42 35531082 A0ABA22B 3B4B9B46 5246DB03 5142855A 69DA2AA8  2EE0000   3FF00F FC000000     1120
Parameter  19:    12BD1 76AB802B 84DB8496 8602A617 12B792B3 90B2A592 2A42AA4B AA2BAAA9 A0D50000   3FF00F FFFC0000     1130
Parameter  20:    2AAC1 AAA8AB52 29530B0F 88AA828A 92C39252 95528512 D392A941 2A682AAC  CF50000   3FF00F FC000000     1140
Parameter  21:    7AA11 2AA86CAB E42B602B F32BA9AA AA8A1A2F  92AA923 292EA82B 2C28A128 83730000   3FF00F FFFE0000     1150
Parameter  22:    CAAD1 6DAA95AE 548C56D7 56545696 56B754B6 A8928B57 A3152D55 35516D40 17CD0000   3FF00F FC020000     1160
Parameter  23:    8AA91 24122D45 A4B534A5 B4AC92A8  B4AAF22 A4AEA10F A562AB46 AAD3AAAB 22EF0000   3FF00F FFFE0000     1170
Parameter  24:    EAAF1 6A8305A3  A9A08AA A8ABA82A B55A95C2 B0CA828C D28B8A8A 8AA922AA 7C8C0000   3FF00F FFFE0000     1180
Parameter  25:    8AA81 A8A92AAB AC8AA953 A853A803 AA93AA8A C102D52B D2AB4AAB 8AA92AA4 EA830000   3FF00F FC000000     1190
Parameter  26:    6AA01 6AAB2F2B  B2A282B 900AD152  3525257 C9D69956 9B428AAD 9A8D5696 2C660000   3FF00F FC020000     11A0
Parameter  27:    74141 AC04AAB6 2A4A2B4B A92B810A 2CA8ACAC A9E52DB5 B195A895 2AAB6AAE 3E740000   3FF00F FC020000     11B0
Parameter  28:    AAAF1 E82A952A 262A161B 920BBA4B B243D056 52569AD2 75DEFC6A 6A69AAAE 16960000   3FF00F FFFC0000     11C0
Parameter  29:    E15F1 105382D7  893C09B 83D3A552 3147892B A1A2A9AB A572A54A 2AABAAAE   9F0000   3FF00F FC000000     11D0
Parameter  30:    82A81 2AAA8A53 A822280F A8CEA4CB B643B66A BE6AADAA 20AAB82B B7A92A27  CAB0000   3FF00F FFFC0000     11E0
Parameter  31:    2AA11 2BABF123 95AE546B D26B2A6B A909390E E9AEA91F 2097B253 B14BA328 9BE30000   3FF00F FC020000     11F0
Parameter  32:    AAA51 6AA28B56 8B57E915 49456D6A 696A206B A12A812B 2126A1AA A02BE8A5 13A30000   3FF00F FC040000     1200
Parameter  33:    CAAA1 6356B724 B568932C 83AC81AC 392AADCA A502A597 A196ABD2 2D532D6F 46FB0000   3FF00F FC000000     1210
Parameter  34:    CAAD1 4AA9B2AA B2AB3960 B56B951B 1419D557 AC07AA12 AA93CA5B 4229EBAF ECF00000   3FF00F FC000000     1220
Parameter  35:     AAB1  FAEC76F 744AD4E3 D4BA968A AA4EAB5A E9626D26 CD9249DA 49282EAD 9CDD0000   3FF00F FC060000     1230
Parameter  36:    2AA11 EAAAAAAA BD4B3153 D552C082 88038923 88ABAAAB AAABC555 95575556 2E7C0000   3FF00F FC000000     1240
Parameter  37:    F4501 BD4A2A0A E86A282A A92A8893 8888AAA9 28A92D55 2950AA55 BAAB2AAB 84E40000   3FF00F FFFE0000     1250
Parameter  38:    8AAD1 AAABAB7A 22DB229A 98D3E2D3 B92AD52A 436A826B 666AE92B 29AAAAA9 D89E0000   3FF00F FFFA0000     1260
Parameter  39:    564B1 D45AD697 50B714AB AA93A8D7 3556B157 8906A8A3 A5A2A5A8 2AAB2AAA BC9B0000   3FF00F FC020000     1270
Parameter  40:    AAAC1 AAAAA84B A84A288A A28A26CB B28AB28B 32CAB60B A46AA929 B42BAAAC A8F00000   3FF00F FFFE0000     1280
Parameter  41:    DAA21 2AA96A63 2A0AC24B F26BBA7A 8BB99BAF A9AEADAE A3AFB2A8 95AAA1AA C80E0000   3FF00F FFFE0000     1290
```

Hình 3.23. Quá trình nạp các parameter

Quá trình khởi tạo parameter neuron (Hình 3.23) thực hiện chính xác. Việc tách 368 bit thành 12 cụm dữ liệu đưa vào mạng neuron đúng với chức năng.

Quá trình khởi tạo neuron instruction (Hình 3.24Hình 3.23) thực hiện đúng, mạng đã có phản hồi.

Tiến hành điều khiển và đưa các packet đầu vào vào mạng. Sau khi nhận được tín hiệu tick_ready thì sẽ đẩy tín hiệu kích hoạt tính toán vào mạng.

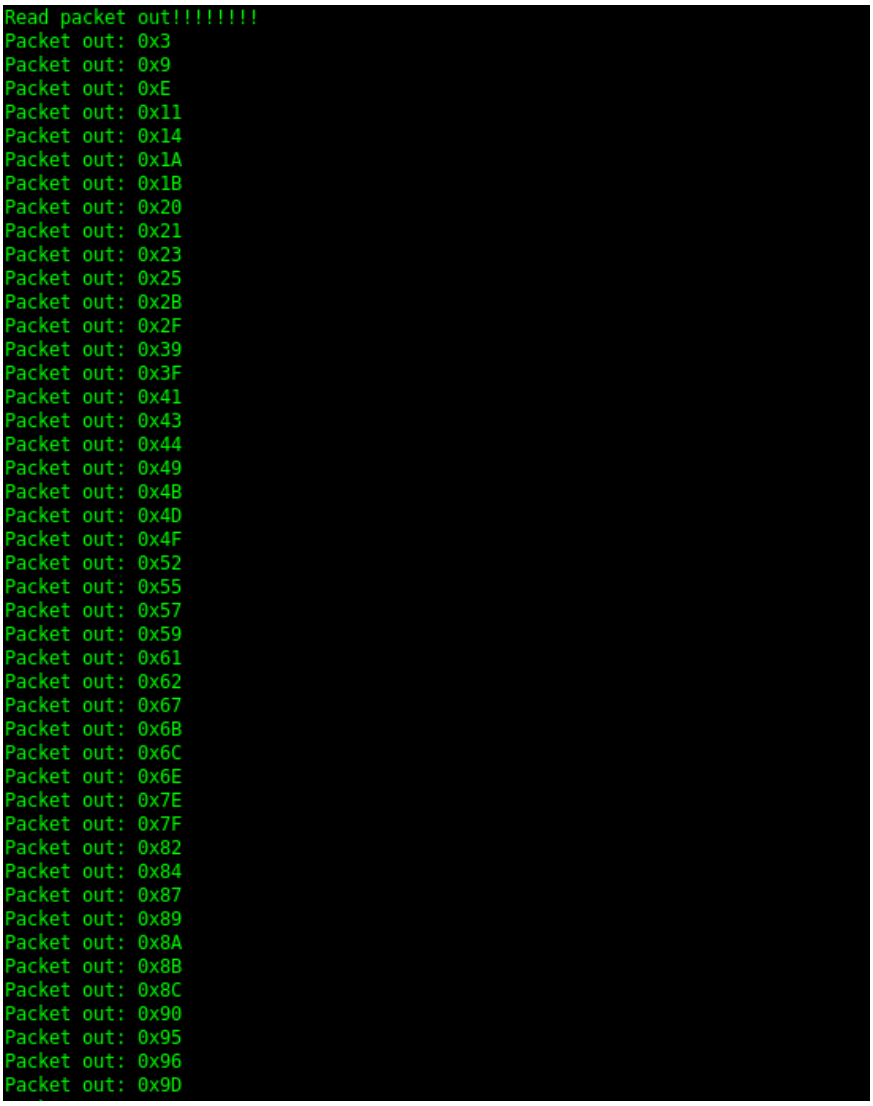
Sau khi có tín hiệu wait_packets, tiến hành đọc các packet đầu ra (Hình 3.26).

```
Neuron instructions   0: 0x0
Neuron instructions   1: 0x1
Neuron instructions   2: 0x0
Neuron instructions   3: 0x1
Neuron instructions   4: 0x0
Neuron instructions   5: 0x1
Neuron instructions   6: 0x0
Neuron instructions   7: 0x1
Neuron instructions   8: 0x0
Neuron instructions   9: 0x1
Neuron instructions  10: 0x0
Neuron instructions  11: 0x1
Neuron instructions  12: 0x0
Neuron instructions  13: 0x1
Neuron instructions  14: 0x0
Neuron instructions  15: 0x1
Neuron instructions  16: 0x0
Neuron instructions  17: 0x1
Neuron instructions  18: 0x0
Neuron instructions  19: 0x1
Neuron instructions  20: 0x0
Neuron instructions  21: 0x1
Neuron instructions  22: 0x0
Neuron instructions  23: 0x1
Neuron instructions  24: 0x0
Neuron instructions  25: 0x1
Neuron instructions  26: 0x0
Neuron instructions  27: 0x1
Neuron instructions  28: 0x0
Neuron instructions  29: 0x1
Neuron instructions  30: 0x0
Neuron instructions  31: 0x1
Neuron instructions  32: 0x0
Neuron instructions  33: 0x1
```

Hình 3.24 Quá trình nạp các instruction

```
Packet in   0: 0x440
Packet in   1: 0x450
Packet in   2: 0x560
Packet in   3: 0x570
Packet in   4: 0x580
Packet in   5: 0x590
Packet in   6: 0x5A0
Packet in   7: 0x5B0
Packet in   8: 0x6B0
Packet in   9: 0x7A0
Packet in  10: 0x7B0
Packet in  11: 0x8A0
Packet in  12: 0x990
Packet in  13: 0xA80
Packet in  14: 0xA90
Packet in  15: 0xB80
Packet in  16: 0xC70
Packet in  17: 0xC80
Packet in  18: 0xD60
Packet in  19: 0xD70
Packet in  20: 0xE60
Packet in  21: 0xE70

Tick ready!!!!!!!!
```

Hình 3.25. Quá trình nạp packet đầu vào



Hình 3.26. Kết quả packet đầu ra

Sau khi kiểm thử trên 10 file đầu vào khác nhau, kết quả cho thấy các packet đầu ra khớp với quá trình mô phỏng

→ Hệ thống thực hiện đúng với các yêu cầu đề ra.

CHƯƠNG 4. KẾT LUẬN

Báo cáo này đã trình bày quá trình thực hiện và triển khai hệ thống SNN lên hệ thống Systemon Chip. Thiết kế đã triển khai và nâng cấp để ghép nối lên SOC và chạy được đúng trên kit KC705. Kết quả demo đã được so sánh với kết quả mô phỏng, cho thấy hệ thống chạy chính xác. Đề tài được viết trên ngôn ngữ verilog và python.

Nhưng do vấn đề về thời gian, em vẫn chưa thể triển khai hệ thống chạy theo thời gian thực sử dụng camera trực tiếp. Trong tương lai, em sẽ tiến hành cải tiến hệ thống, hoàn thiện để ghép nối được camera và hiện thị kết quả sau khi khảo sát lên màn hình.

TÀI LIỆU THAM KHẢO

[1] F. Akopyan, "TrueNorth: Design and Tool Flow of a 65mW".

[2] R. P. K. R. M. I. E. R. S. V. Joshua Mack, "RANC: Reconfigurable Architecture for," 01 11 2020.

[3] Enjoy Digital, [Online]. Available: https://github.com/enjoy-digital/litex. [Accessed 8 August 2022].

[4] dprete, "Asynchronous dual clock FIFO," 2014. [Online]. Available: https://github.com/dpretet/async_fifo.

[5] nguyenviethi, "EDABK Litex," Github, September 2022. [Online]. Available: https://github.com/nguyenvietthi/edabk_litex/tree/main/soc_snn.