TRƯỜNG ĐẠI HỌC BÁCH KHOA TP HỒ CHÍ MINH

Báo cáo thực tập tốt nghiệp
Xây dựng mô hình neural network trên FPGA
Neural network implemented on FPGA

Nguyễn Thanh Toàn
MSSV: 2014777
toan.nguyenpsc@hcmut.edu.vn

Khoa Điện-Điện tử
Chuyên ngành điện tử viễn thông

Giảng viên hướng dẫn: **Trần Hoàng Quân**

Bộ môn: Điện tử
Khoa: Điện-Điện tử

TP HỒ CHÍ MINH, 7/2023

Mục lục

PHẦN MỞ ĐẦU

1. Tính cấp thiết, ý nghĩa thực tiễn của đề tài

Ngày nay với sự phát triển nhanh chống của công nghệ trí tuệ nhân tạo, đặc biệt là các ứng dụng liên quan đến máy học như phân loại ảnh, giám sát hình ảnh, nhận diện giọng nói,... Tấc cả các ứng dụng này phần lớn thực hiện được nhờ vào mô hình neural network, phép toán quan trọng nhất trong mô hình neural networt là phép toán tích chập (convolution). Tuy nhiên phép toàn này bao gồm nhiều phép nhân và phép cộng. Vì vậy với một bộ xữ lý trung tâm (CPU) thực hiện các phép toán một cách tuần tự thì việc thực hiện các phép tích chập sẽ mất khá nhiều thời gian, Đối với lĩnh vực cần nguồn dữ liệu lớn như máy học thì việc tính toán tuần tự như CPU sẽ là một trở ngại lớn. Vì vậy chúng ta cần một bộ xữ lý tính toán có thể tính toán song song nhiều phép tính cùng lúc và có tốc độ tính toán cao để đáp ứng nhu cầu của các ứng dụng máy học. FPGA có thể đáp ứng được nhu cầu về tính toán song song và tốc độ xữ lý cao của các ứng dụng liên quan đến máy học như mạng thần kinh ảo, nhận diện hình ảnh, nhận diện khuôn mặt, phát hiện vật thể.

2. Nhiệm vụ đồ án

Xây dựng bộ tính toán song song với tốc độ xữ lý cao trên FPGA, cụ thể hơn là bộ FPGA DE2-155.

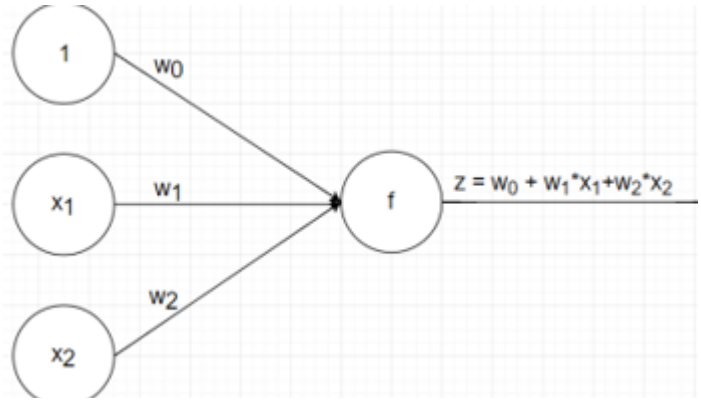
3. Lĩnh vực ứng dụng:

Hệ thống nhúng, các ứng dụng cần các thuật toán liên quan máy học, tích chập.

PHẦN 1: PHÉP TOÁN TÍCH CHẬP

1. Giới thiệu về phép toán tích chập

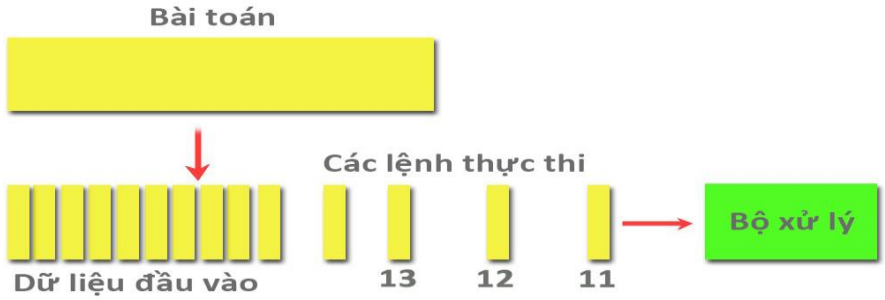
Phép toán tích chập là một trong những phép toán quan trọng nhất của thuật toán máy học nói chung hay trong neural network nói riêng. Về cơ bản phép toán tích chập là tổng của các phép toán nhân.



Hình 1: Minh họa phép toán tích chập

Phép toán tích chập có thể được hiểu như sau: Đầu vào của phép toán bao gồm các input và các weight, Mỗi một input sẽ có một weight tương ứng với input đó và ngõ ra sẽ được tính toán theo công thức $y = x_0 \times w_0 + x_1 \times w_1 + x_2 \times w_2 + ... + x_n \times w_n$.

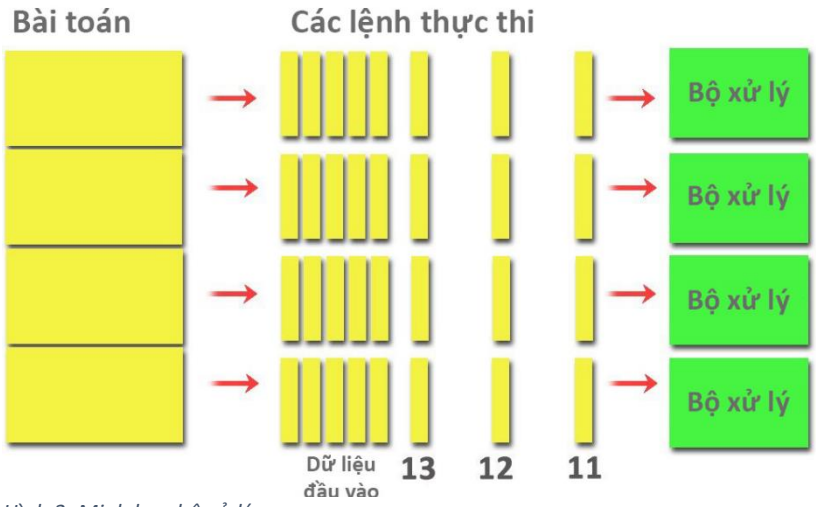
Đối với bộ xử lý tuần tự thì phép toán trên sẽ được chia nhỏ và thực hiện lần lượt bằng cách thực hiện $x_0 \times w_0$ sẽ lưu tạp kết quả vào một thanh ghi và tiếp tục thực hiện $x_1 \times w_1$, rồi tiếp cộng kết quả này cho thanh ghi chứa kết quả $x_0 \times w_0$ đã được tính lần trước, cứ như vậy cho đến khi thực hiện xong phép toán, như vậy ta có thể thấy rằng một bộ xữ lý tuần tự vẫn hoàn toán có khã năng thực hiện chính xác phép toán tích chập, tuy nhiên phép toán được chia ra thành rất nhiều phép toán nhỏ hơn, mỗi chu kỳ thực hiện tính toán của bộ xử lý chỉ thực hiện duy nhất một phép toán nhỏ, phép toán nhỏ tiếp theo cần phép chờ bộ xử lý thực hiện xong phép toán trước đó thì phép toán sau mới được thực hiện.



Hình 2: Minh họa bộ xử lý tuần tự

Đối với bộ xử lý song song sẽ có phương pháp thực hiện hoàn toán khác so với bộ xử lý tuần tự. Bộ xử lý song song sẽ thực hiện phép toán $x_0 \times w_0$ tại một đơn vị tính toán và thực hiện phép toán $x_1 \times w_1$ tại một đơn vị tính toán khác mà không phải chờ phép toán $x_0 \times w_0$ thực hiện xong, cứ như vậy cho các phép nhân còn lại. Lợi thế của việc thực hiện tính toán song song là quá rõ ràng trong trường hợp này, Các phép toán được hiện song song tại cùng một thời điểm. Điều này sẽ tiết kiệm được rất nhiều thời gian cho bộ xữ lý khi thực hiện các phép toán tích chập.

2. **Node neural**



Hình 3: Minh họa bộ xử lý song song

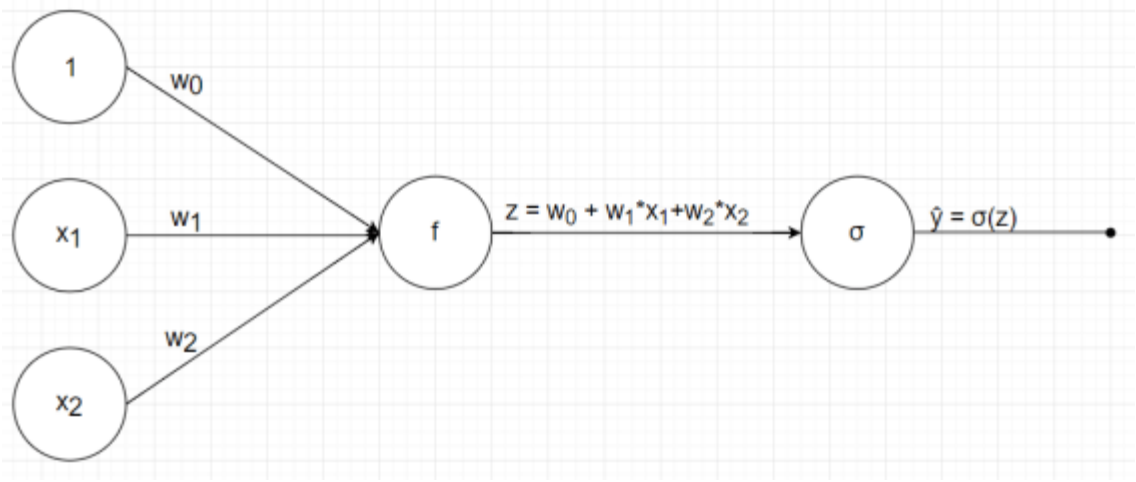
Logistic regression là mô hình neural network đơn giản nhất chỉ với input layer và output layer. Mô hình của logistic regression $\tilde{y} = \sigma(w_0 \times 1 + w_1 \times x_1 + w_2 \times x_2 + ... + w_n \times x_n)$

Có 2 bước để thực hiện mô hình trên bao gồm: Bước tính tổng linear

$$z = w_0 \times 1 + w_1 \times x_1 + w_2 \times x_2 + ... + w_n \times x_n$$

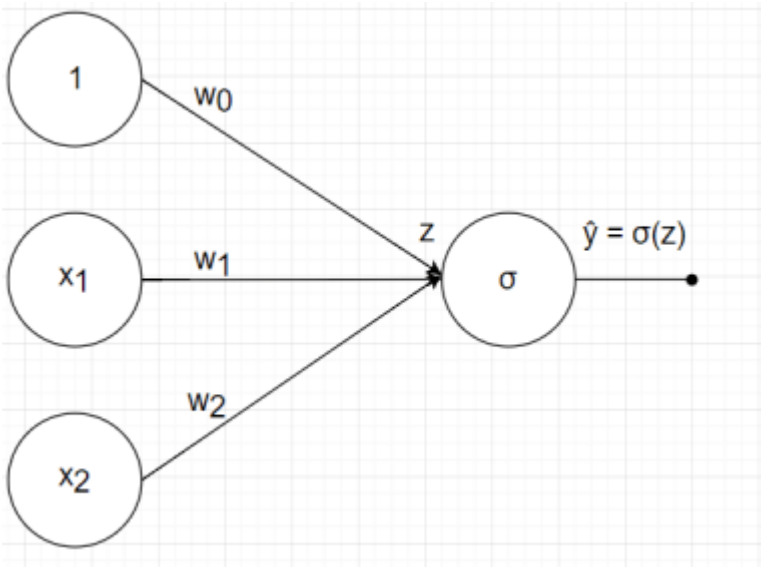
Và phần áp dụng sigmoid function:

$$\tilde{y} = \sigma(z)$$



Hình 4: Minh họa một neural trong neural network

Để biểu diễn gọn lại ta sẽ hiểu ngầm kết quả sau khi tính toán qua bước một tính tổng linear thì sẽ cho kết quả này áp dụng hàm activation.



Hình 5: Minh họa neural của neural network dạng rút gọn

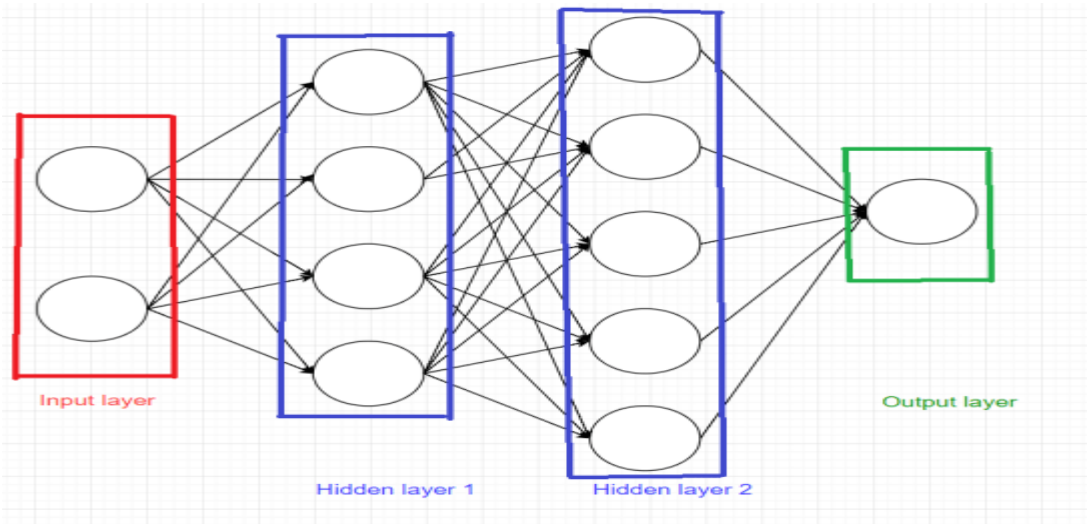
Trong đó có một trọng số đặc biệt mà ngõ vào luôn luôn bằng 1, ta sẽ gọi trọng số đặc biệt này là hệ số bias. Khi tính toán luôn được thêm 1 để tính hệ số bias. phương trình đường thẳng sẽ thế nào nếu bỏ qua hệ số bias, phương trình lúc này có dạng: $w1*x+w2*y=0$, đối với đồ thị của phương trình này sẽ luôn đi qua gốc tọa độ và nó không tổng quát

hóa phương trình đường thẳng, nên ta có thể không tìm được phương trình mong muốn để biểu diễn được tập dữ liệu, vì vậy ta phải cần thêm vào hệ số bias để tổng quát hóa phương trình để biểu diễn tốt hơn các tập dữ liệu trong thực tế.

Bước thứ hai của neural là áp dụng hàm activation (activation function) là một bước rất quan trọng. Nếu không có bước này thì phép biến đổi không có tính chất phi tuyến, việc này không khác gì chúng ta thêm một tầng ẩn nữa vì phép biến đổi cũng chỉ đơn thuần là nhân đầu ra với các weights. Với chỉ những phép tính đơn thuần như vậy, trên thực tế mạng neural sẽ không thể phát hiện ra những quan hệ phức tạp của dữ liệu (ví dụ như: dự đoán chứng khoán, các bài toán xử lý ảnh hay các bài toán phát hiện ngữ nghĩa của các câu trong văn bản). Nói cách khác nếu không có các activation functions, khả năng dự đoán của mạng neural sẽ bị giới hạn và giảm đi rất nhiều, sự kết hợp của các activation functions giữa các tầng ẩn là để giúp mô hình học được các quan hệ phi tuyến phức tạp tiềm ẩn trong dữ liệu.

3. **Mô hình neural network**

Mô hình tổng quát của mô hình neural network giả sử mô hình gồm 1 lớp input, 1 lớp output, 2 lớp hidden.



Hình 6: Minh họa mô hình neural network

Layer đầu tiên là input layer, các layer ở giữa được gọi là hidden layer, layer cuối cùng được gọi là output layer. Các hình tròn được gọi là node.

Mỗi mô hình luôn có 1 input layer, 1 output layer, có thể có hoặc không các hidden layer. Tổng số layer trong mô hình được quy ước là số layer – 1 (Không tính input layer). Ví dụ như ở hình trên có 1 input layer, 2 hidden layer và 1 output layer. Số lượng layer của mô hình là 3 layer.

Mỗi node trong hidden layer và output layer liên kết với tất cả các node ở layer trước đó với các hệ số w riêng, mỗi node có 1 hệ số bias b riêng, diễn ra 2 bước: tính tổng linear và áp dụng activation function.

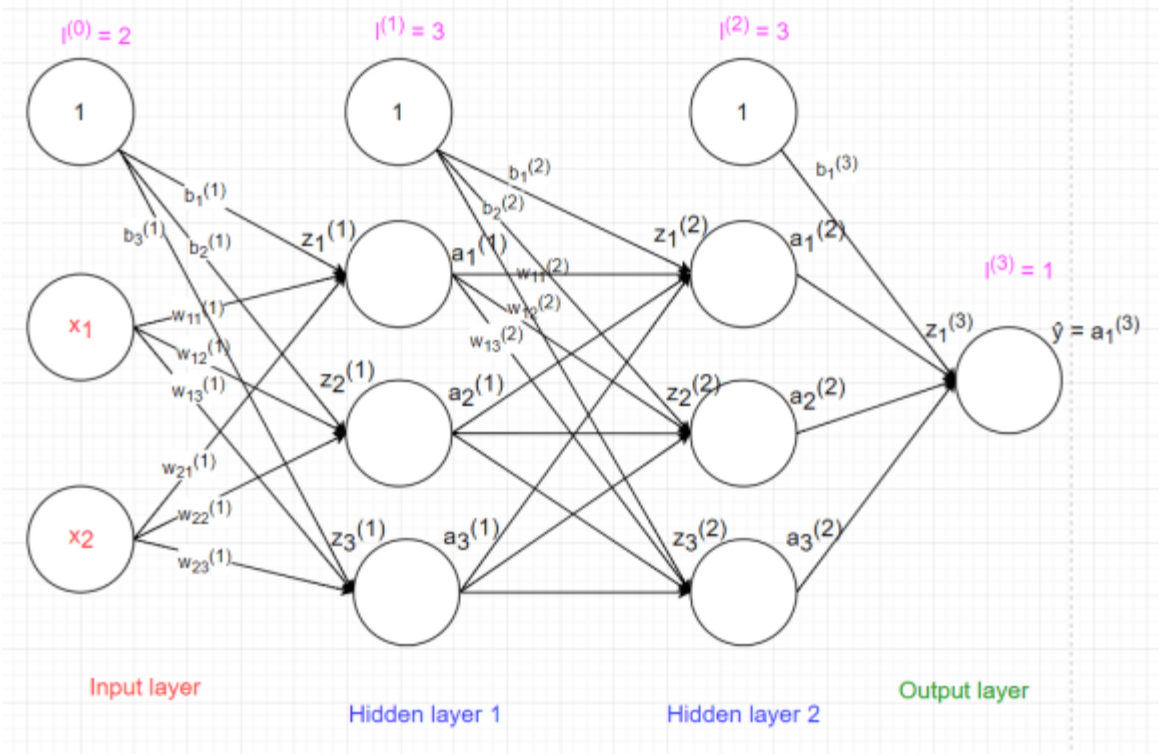
Kí hiệu:

Số node trong hidden layer thứ i là $l(i)$.

Ma trận $W(k)$ kích thước $l(k-1)*l(k)$ là ma trận hệ số giữa layer (k-1) và layer k, trong đó $wij(k)$ là hệ số kết nối từ node thứ i của layer k-1 đến node thứ j của layer k.

Vector $b(k)$ kích thước $lk*1$ là hệ số bias của các node trong layer k, trong đó $bi(k)$ là bias của node thứ i trong layer k.

Với node thứ i trong layer l có bias $bi(l)$ thực hiện 2 bước, bao gồm tính tổng linear: $zi(l)=\sum j=1 l(l-1)aj(l-1)*wji(l)+bi(l)$, là tổng tất cả các node trong layer trước nhân với hệ số w tương ứng, rồi cộng với bias b và áp dụng activation function: $ai(l)=\sigma(zi(l))$



Hình 7: Mô hình chi tiết neural network fully connected

Vector $z(k)$ kích thước $l(k)*1$ là giá trị các node trong layer k sau bước tính tổng linear, và vector $a(k)$ kích thước $l(k)*1$ là giá trị của các node trong layer k sau khi áp dụng hàm activation function.

Mô hình neural network trên gồm 3 layer. Input layer có 2 node $l(0)=2$), hidden layer 1 có 3 node, hidden layer 2 có 3 node và output layer có 1 node.

Do mỗi node trong hidden layer và output layer đều có bias nên trong input layer và hidden layer cần thêm node 1 để tính bias (nhưng không tính vào tổng số node layer có).

Tại node thứ 2 ở layer 1, ta có:

$$z2(1)=x1*w12(1)+x2*w22(1)+b2(1)$$

$$a2(1)=\sigma(z2(1))$$

Hay ở node thứ 3 layer 2, ta có:

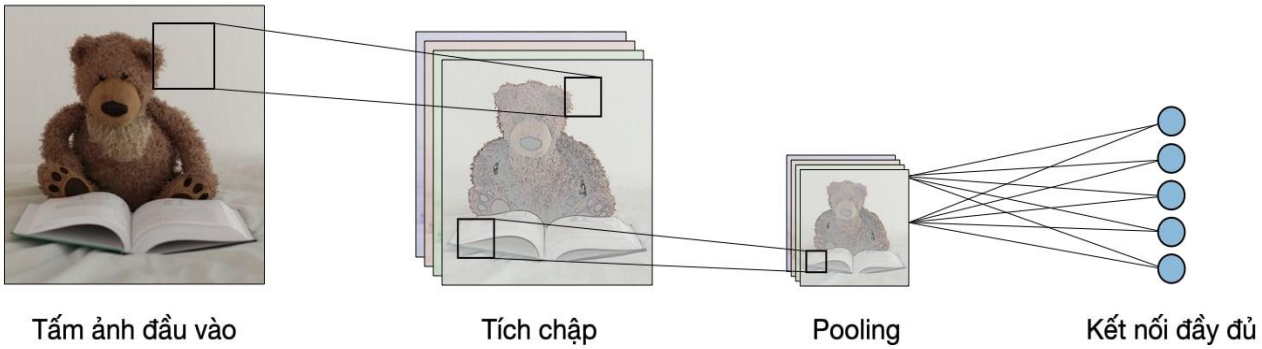
$$z3(2)=a1(1)*w13(2)+a2(1)*w23(2)+a3(1)*w33(2)+b3(2)$$

$$a3(2)=\sigma(z3(2))$$

PHẦN 2: MÔ HÌNH CNN (CONVOLUTION NEURAL NETWORK)

1. Kiến trúc mô hình CNN – convolution neural network

Mạng neural tích chập (Convolutional neural networks), còn được biết đến với tên CNNs, là một dạng mạng neural được cấu thành bởi các tầng sau:



Hình 8: Minh họa mô hình cnn - convolution neural network

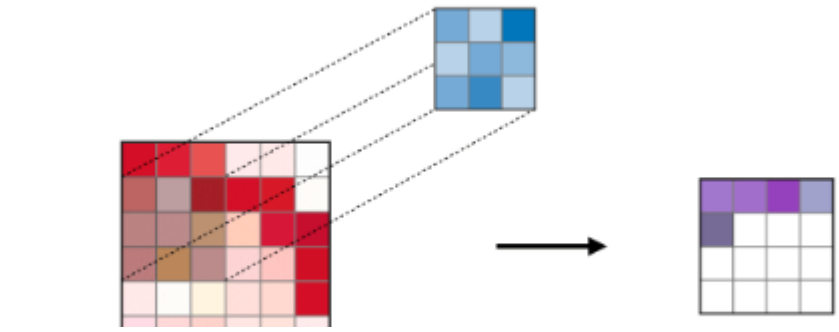
Tầng tích chập và tầng pooling là hai tầng đặc trưng và quan trọng nhất của mô hình cnn – convolution neural network.

2. Các tầng của mô hình CNN.

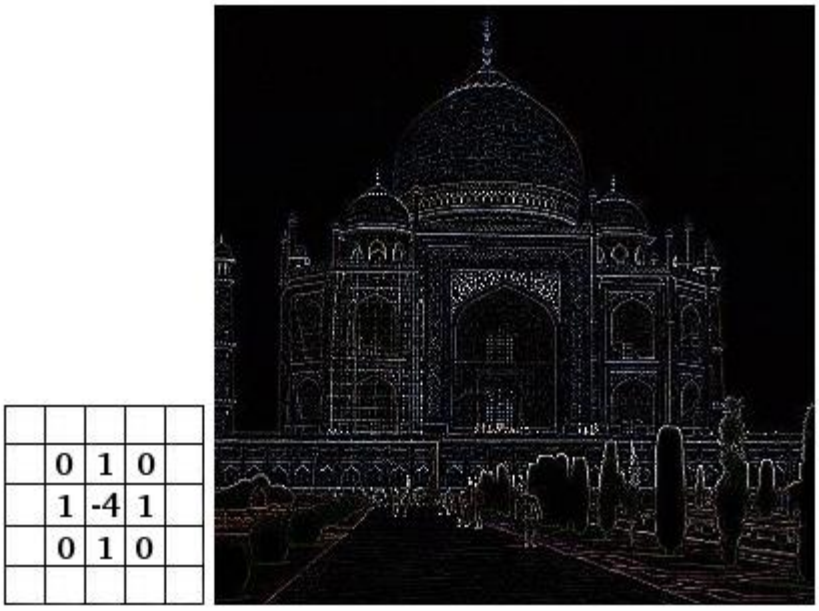
2.1 Tầng tích chập (CONV).

Tầng tích chập (CONV) sử dụng các bộ lọc để thực hiện phép tích chập khi đưa chúng đi qua đầu vào I theo các chiều của nó. Các siêu tham số của các bộ lọc này bao gồm kích thước bộ lọc F và độ trượt (stride) S. Kết quả đầu ra O được gọi là feature map hay activation map. Đây là lớp đặc trưng chỉ có ở mô hình CNN – Convolution neural network

Bước tích chập cũng có thể được khái quát hóa cả với trường hợp một chiều (1D) và ba chiều (3D)



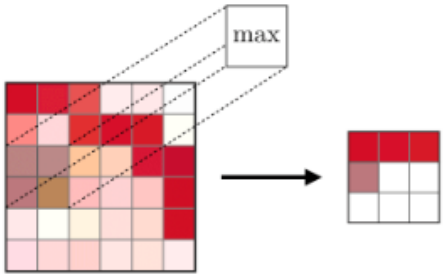
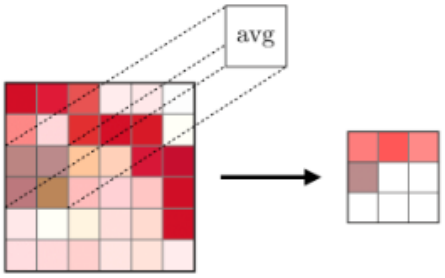
Hình 9: Minh họa thuật toán lớp tích chập



Hình 10: Minh họa ví dụ output lớp tích chập

2.2 Pooling (POOL).

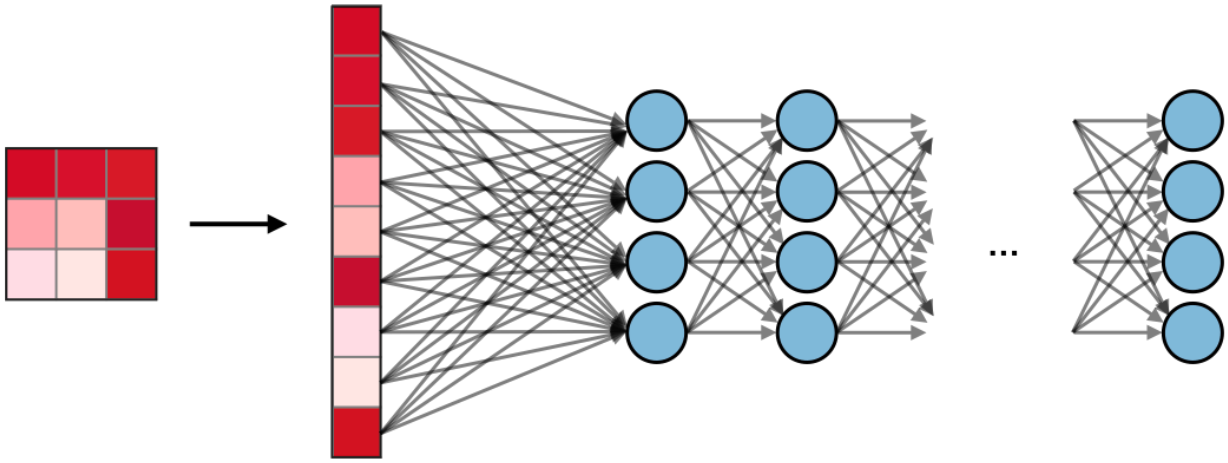
Tầng pooling (POOL) là một phép downsampling, thường được sử dụng sau tầng tích chập, giúp tăng tính bất biến không gian. Cụ thể, max pooling và average pooling là những dạng pooling đặc biệt, mà tương ứng là trong đó giá trị lớn nhất và giá trị trung bình được lấy ra.

Kiểu	Max pooling	Average pooling
Chức năng	Từng phép pooling chọn giá trị lớn nhất trong khu vực mà nó đang được áp dụng	Từng phép pooling tính trung bình các giá trị trong khu vực mà nó đang được áp dụng
Minh họa		
Nhận xét	• Bảo toàn các đặc trưng đã phát hiện • Được sử dụng thường xuyên	• Giảm kích thước feature map • Được sử dụng trong mạng LeNet

Hình 11: Phân biệt hai loại phép toán pooling

2.3 Fully Connected (FC).

Tầng kết nối đầy đủ (FC) nhận đầu vào là các dữ liệu đã được làm phẳng, mà mỗi đầu vào đó được kết nối đến tất cả neuron. Trong mô hình mạng CNNs, các tầng kết nối đầy đủ thường được tìm thấy ở cuối mạng và được dùng để tối ưu hóa mục tiêu của mạng ví dụ như độ chính xác của lớp.

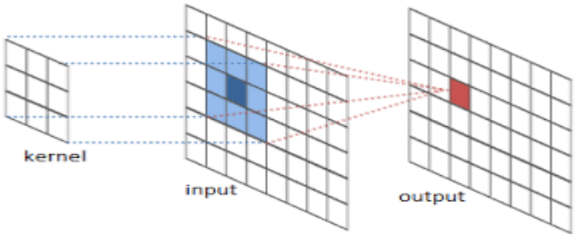


Hình 12: Minh họa lớp fully connected

PHẦN 3: THỰC HIỆN MÔ HÌNH CNN TRÊN FPGA DE2

1. Thực hiện convolution layer

Lớp tích chập hay convolution layer là một trong những đặc trưng cơ bản của mô hình mạng neural thần kinh ảo tích chập – cnn (Convolution neural network). Đồng thời cũng là lớp quan trọng nhất trong mô hình CNN. Khác với mạng ANN, lớp ẩn (hidden layer) đầu tiên sẽ luôn luôn kết nối với toàn bộ dữ liệu đầu vào, trong khi đó đối với mô hình CNN, input map sẽ được chia ra thành những vùng nhỏ, và các vùng thường sẽ chồng lấn lên nhau và được nhân với một ma trận kernal và đầu ra là một feature map. Mục đích của việc này là làm nỗi bậc nên những đặc trưng của input map.



Hình 13: Minh họa hoạt động của lớp tích chập (convolution layer)

Kích thước của output feature map được xác định như sau: Giả sử input map có kích thường chiều rộng là w và chiều cao là h như vậy kích thước của input map là $(w \times h)$. Tốc độ dịch ma trận kernal (stride) s =1 tức dịch chuyển kernal 1 pixel mỗi lần tính toán, padding – phần được thêm vào xung quanh input map nhằm tăng kích thước cho input map, giả sử padding start (padding được thêm vào phần trước của input map) và padding end (padding được thêm vào phần cuối của input map) đều bằng 0. Như vậy ra có thể xác định kích thước của output feature map theo công thức sau:

$$w_{outputFeaterMap} = \frac{w-k}{s} + 1$$

$$h_{outputFeatureMap} = \frac{h-k}{s} + 1$$

Trong đó :

$w_{outputFeatureMap}$: Chiều rộng của output feature map (pixel)

$h_{outputFeatureMap}$: Chiều cao của ouput feature map (pixel)

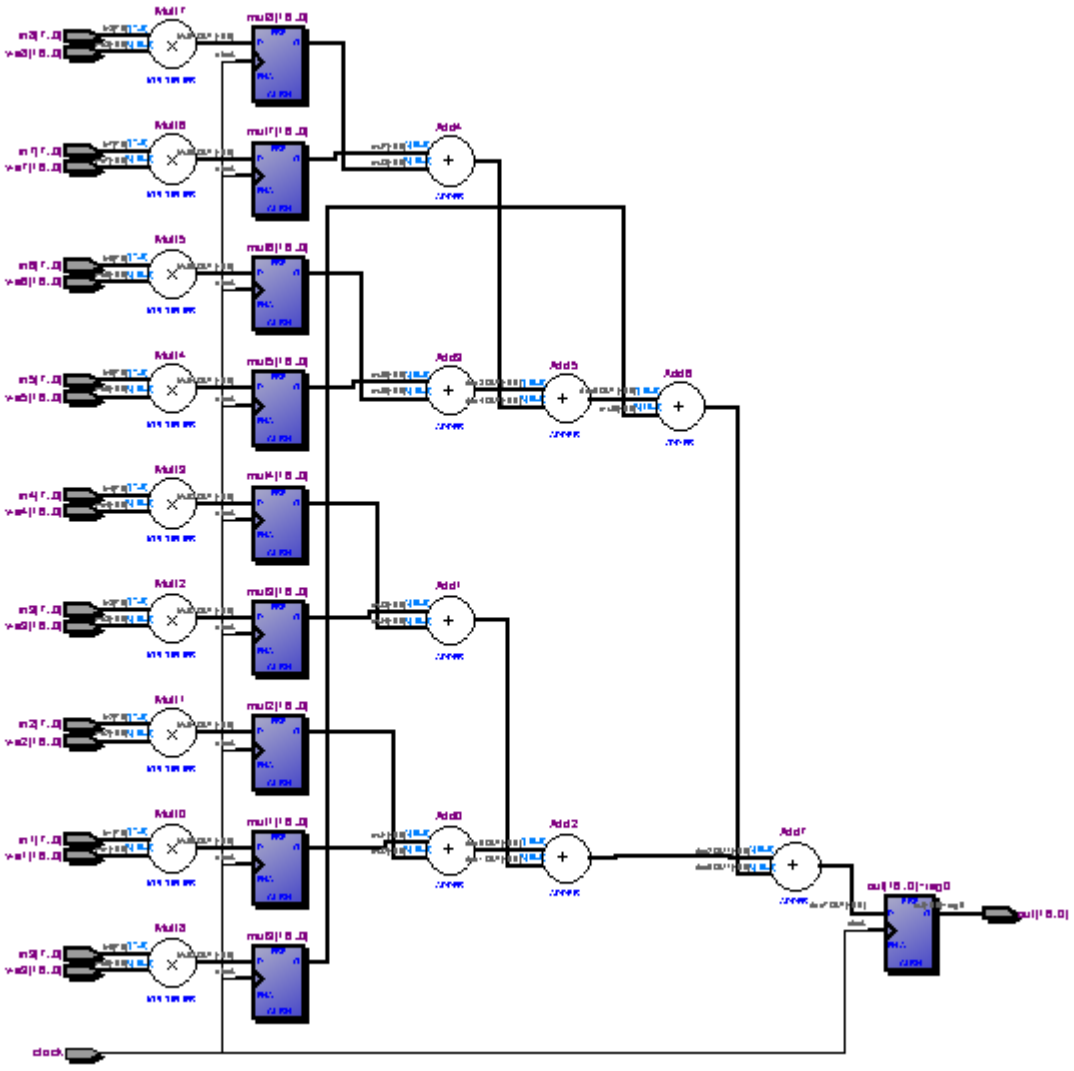
w : Chiều rộng của input map (pixel)

h : Chiều cao của input map (pixel)

k : Kích thước của ma trận kernal

1.1 Thiết kết core computing kernal

Core computing kernal là một đơn vị tính toán có ngõ vào là 9 input của input map và 9 trọng số của ma trận kernal, ngõ ra là giá trị tích chập của 9 input map và 9 trọng số kernal nói trên.



Hình 14: Block diagram core computing kernal

Như thiết kế các ngõ vào input từ 1 đến 9 sẽ được nhân tương ứng với trọng số đầu vào từ 1 đến 9. Input của input map là số nguyên 8 bit, để tránh bị tràn dẫn đến kết quả tính toán bị sai thì ta chọn trọng số ngõ vào có 17 bit gồm 1 bit MSB là bit dấu vì trọng số của ma trận kernal có thể âm hoặc dương hoặc bằng 0. Kết quả của core computing kernal là một số cũng có 17 bit với bit MSB là bit dấu.

Ngoài ra để tăng thêm tốc độ tính toán – tăng tần số hoạt động lên cao thì trong thiết kế được thêm vào 2 lớp data buffer computing (bộ nhớ đệm) lớp thứ nhất phía sau các bộ nhân và lớp thứ 2 phía sau các bộ cộng. Đồng thời các bộ cộng cũng được sắp xếp để có số tầng thực hiện bộ cộng là thấp nhất là 4 tầng, góp phần làm giảm thời gian delay. Các bộ đệm

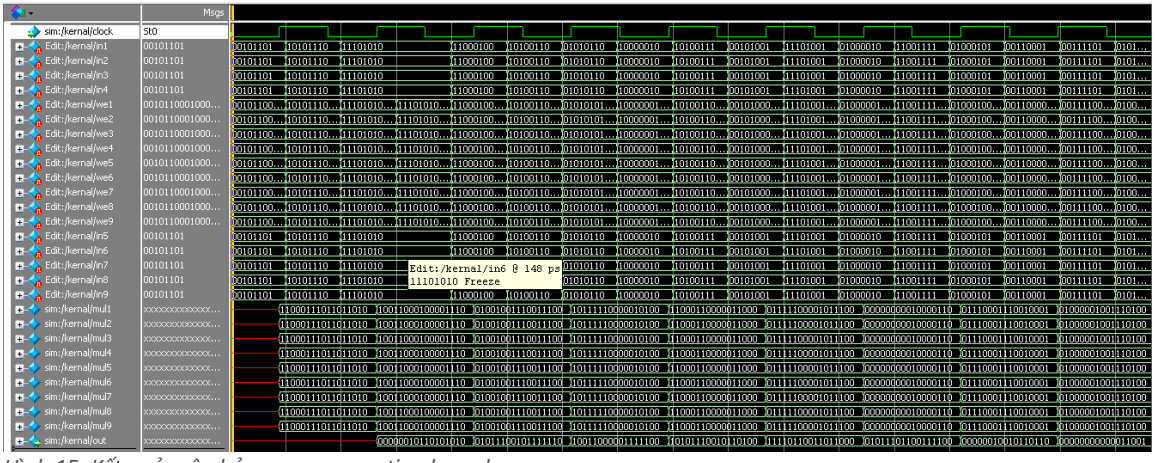
này đồng bộ với nhau về xung clock, Vì vậy data delay buffer output (Độ trễ bộ đệm) của module này là 2 chu kỳ xung clock.

Source code verilog của module core computing kernal được đặt tên module là "kernla" và được đính kèm bên dưới.

```
module kernal(clock,in1, in2, in3, in4, in5, in6, in7, in8, in9,we1, we2, we3, we4, we5, we6, we7, we8, we9, out);
	input clock;
	input [7:0] in1, in2, in3, in4, in5, in6, in7, in8, in9;
	input [16:0] we1, we2, we3, we4, we5, we6, we7, we8, we9;
	output reg [16:0] out;
	reg [16:0] mul1, mul2, mul3, mul4, mul5, mul6, mul7, mul8, mul9;
	always @(posedge clock)
	begin
		mul1 <= in1*we1;
		mul2 <= in2*we2;
		mul3 <= in3*we3;
		mul4 <= in4*we4;
		mul5 <= in5*we5;
		mul6 <= in6*we6;
		mul7 <= in7*we7;
		mul8 <= in8*we8;
		mul9 <= in9*we9;
		out = ((mul1 + mul2) + (mul3 + mul4)) + (((mul5 + mul6) + (mul7 + mul8)) + mul9);
	end
endmodule
```

Kết quả mô phỏng core computing kernal.

Từ kết quả mô phỏng trên ta có thể chứng minh được delay data buffer output của

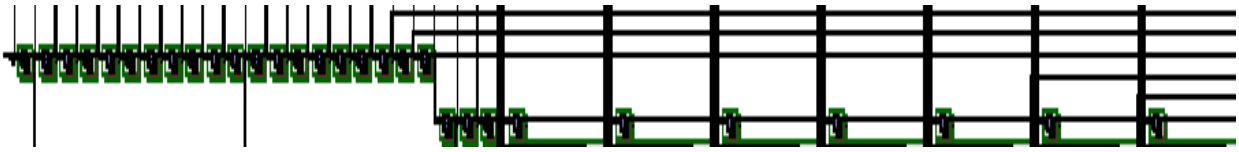


Hình 15: Kết quả mô phỏng core computing kernal

module là 2 xung clock.

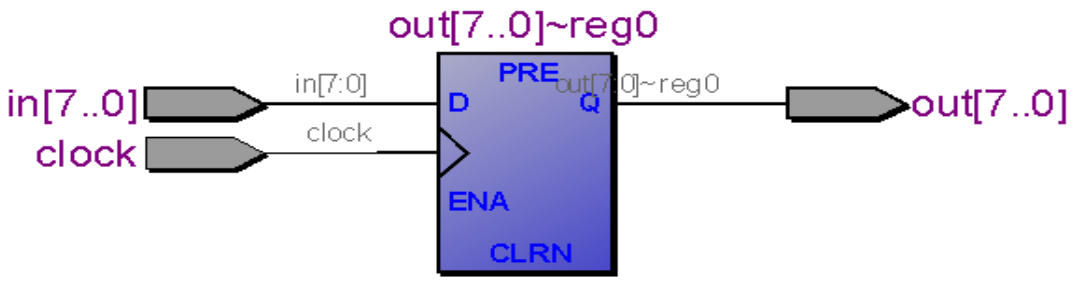
1.2 Thiết kế bộ nhớ đệm cho data input (buffer input)

Giả sử kích thước của ma trận kernal là 3x3 tức k = 3, và kích thước của input map là 10x10 tức w = 10, h = 10. Thông thường input sẽ được lấy từ một camera vậy nên input map sẽ là một chuỗi dữ liệu đầu vào có clock pixel - PCLK (báo hiệu hoàn thành truyền một pixel), tín hiệu đồng bộ kết thúc hàng – HREF (báo hiệu kết thúc một frame, chuyển đến hàng tiếp theo), tín hiệu đồng bộ kết thúc – VSYNK (báo hiệu kết thúc input map, quay lại ô đầu tiên). Dựa vào các tín hiệu phía trên ta sẽ xây dựng một module có tên là bufferInput để chứa tạm các input của input đồng thời trở thành đầu vào cho core computing kernal.



Hình 16: Block diagram của buffer data input

Các ô nhớ đệm sẽ được sắp xếp trải dài từ 29 đến 0, ngõ vào được nối với ô nhớ 29, tín hiệu xung clock là tín hiệu xung PCLK (báo hiệu truyền xong 1 pixel) cứ như vậy sau 30 xung PCLK thì dữ liệu sẽ được nạp đầy vào bộ buffer theo đúng thứ tự từ 0 đến 29 tương ứng với 3 frame, lúc này khi xung HREF được active thì 30 ô nhớ đệm này sẽ được đưa vào đầu vào của bộ core computing kernal để thực hiện tính toán.



Hình 17: Cấu trúc bên trong của 1 ô nhớ buffer

Source code verilog của bộ nhớ đệm input map:

```
// On the top entity
```

```
wire [7:0] b [0:29];
buff node0(inp, b[29], clk);
buff node1(b[29], b[28],clk);
buff node2(b[28], b[27],clk);
buff node3(b[27], b[26],clk);
buff node4(b[26], b[25],clk);
buff node5(b[25], b[24],clk);
buff node6(b[24], b[23],clk);
buff node7(b[23], b[22],clk);
buff node8(b[22], b[21],clk);
buff node9(b[21], b[20],clk);
buff node10(b[20], b[19],clk);
buff node11(b[19], b[18],clk);
buff node12(b[18], b[17],clk);
buff node13(b[17], b[16],clk);
buff node14(b[16], b[15],clk);
buff node15(b[15], b[14],clk);
buff node16(b[14], b[13],clk);
buff node17(b[13], b[12],clk);
buff node18(b[12], b[11],clk);
buff node19(b[11], b[10],clk);
buff node20(b[10], b[9],clk);
buff node21(b[9], b[8],clk);
buff node22(b[8], b[7],clk);
buff node23(b[7], b[6],clk);
buff node24(b[6], b[5],clk);
buff node25(b[5], b[4],clk);
buff node26(b[4], b[3],clk);
buff node27(b[3], b[2],clk);
buff node28(b[2], b[1],clk);
buff node29(b[1], b[0],clk);
//======================
module buff (in, out, clock);
input clock;
input [7:0] in;
output reg [7:0] out;
```

```
always @(posedge clock) begin
        out <= in;
end
endmodule
```

Sau khi đã có bộ nhớ đệm input map, ta chỉ cần kết nối ngõ ra của bộ đệm vào đầu vào của core computing kernal là hoàn thành.

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29

Hình 19: Các ô nhớ là đầu vào core computing kernal thứ nhất

Tương tự như vậy thì các ô nhớ sẽ là ngõ vào của core computing kernal thứ hai là:

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29

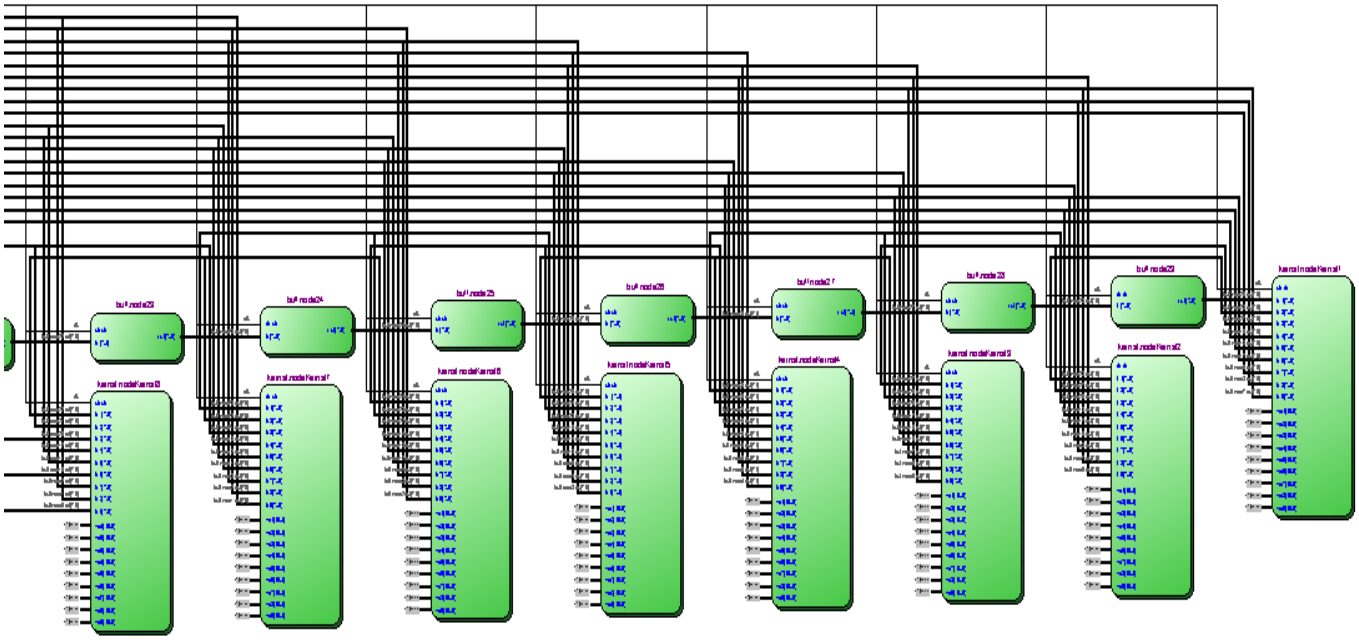
Hình 18: Các ô nhớ là đầu vào core compuing thứ hai

Tương tự như vậy cho các core computing còn lại. Như đã giả sử từ trước input map có kích thước 10x10 (w=10, h=10), kernal có kích thước 3x3 (k=3), hệ số dịch chuyển kernal hay stride là 1 (s=1) Dựa vào công thức ta trình bày phía trên ta tính được kích thước của ouput feature map là:

$$w_{outputFeatureMap} = \frac{w\text{-}k}{s} + 1 = \frac{10-3}{1} + 1 = 8$$

$$h_{outputFeatureMap} = \frac{h\text{-}k}{s} + 1 = \frac{10-3}{1} + 1 = 8$$

Như vậy kích thước của output feature map là 8x8. Vậy mỗi frame sẽ cần 8 core computing kernal unit tính toán song song.



Hình 20: Kết nối bộ nhớ đệm đến core computing kernal

Source code verilog kết nối giữa buffer input map và core computing kernal (ghi chú: Tên của buffer input map được đặt là b[n] với n là số thứ tự của ô buffer từ 0 đến 29 ô nhớ)

```
//=================KetNoiTuBufferDenCoreComputingKernal============
kernal nodeKernal1(href, b[0], b[1], b[2], b[10], b[11], b[12], b[20], b[21], b[22],w1, w2, w3, w4, w5, w6, w7, w8, w9);
kernal nodeKernal2(href,b[1],b[2],b[3],b[11],b[12],b[13],b[21],b[22],b[23],w1, w2, w3, w4, w5, w6, w7, w8, w9);
kernal nodeKernal3(href,b[2],b[3],b[4],b[12],b[13],b[14],b[22],b[23],b[24],w1, w2, w3, w4, w5, w6, w7, w8, w9);
kernal nodeKernal4(href,b[3],b[4],b[5],b[13],b[14],b[15],b[23],b[24],b[25],w1, w2, w3, w4, w5, w6, w7, w8, w9);
kernal nodeKernal5(href,b[4],b[5],b[6],b[14],b[15],b[16],b[24],b[25],b[26],w1, w2, w3, w4, w5, w6, w7, w8, w9);
kernal nodeKernal6(href,b[5],b[6],b[7],b[15],b[16],b[17],b[25],b[26],b[27],w1, w2, w3, w4, w5, w6, w7, w8, w9);
kernal nodeKernal7(href,b[6],b[7],b[8],b[16],b[17],b[18],b[26],b[27],b[28],w1, w2, w3, w4, w5, w6, w7, w8, w9);
kernal nodeKernal8(href,b[7],b[8],b[9],b[17],b[18],b[19],b[27],b[28],b[29],w1, w2, w3, w4, w5, w6, w7, w8, w9);
```

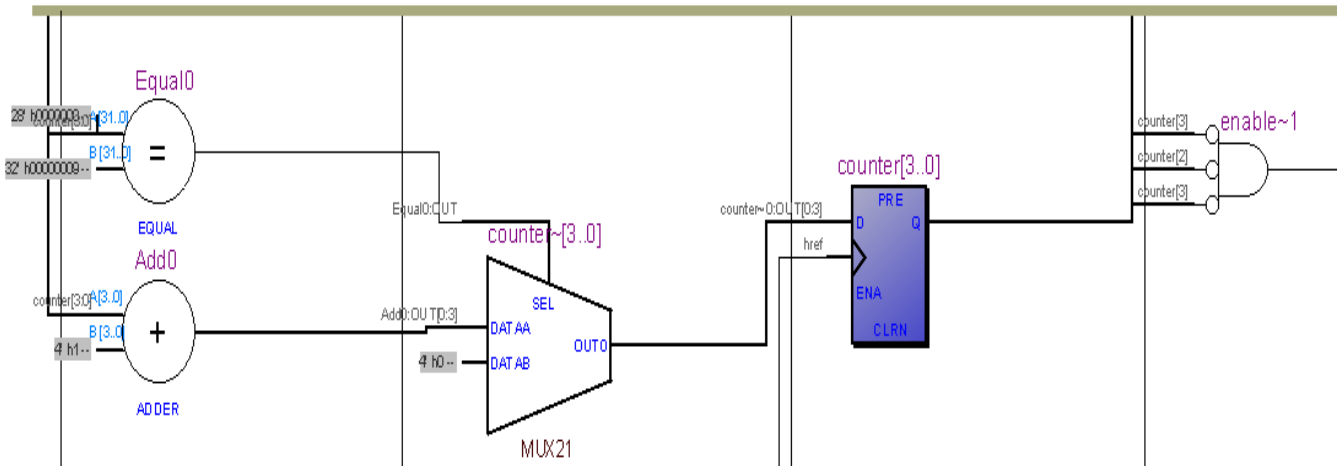
1.3 Thiết kế bộ đệm output feature map của convolution layer.

Bộ đệm hay bộ nhớ của phần output feature map của convolution layer dùng để lưu trữ đầu ra của convolution layer đồng thời cũng là đầu vào cho convolution layer tiếp theo, Tiếp tục xem xét giả sử về mô hình cũng chúng ta có input map 10x10, kích thước của kernal của convolation layer thứ nhất là 3, stride là 1. Như vậy output feature map sẽ có kích thước là 8x8. Như vậy ta sẽ thực hiện thiết kế theo mô hình bộ nhớ như sau:

index								
1	1	2	3	4	5	6	7	8
2	9	10	11	12	13	14	15	16
3	17	18	19	20	21	22	23	24

Hình 21: Mô hình bộ nhớ của output feature map layer1

Tuy nhiên nếu xem xét ở lớp buffer input map, ta có 2 xung clock chứ giá trị rác vì chưa nạp đủ các frame vào bộ buffer (Xem chi tiết tại hình 19) Vì vậy ta cần thiết kế thêm 2 ô nhớ đệm và một biến đếm bỏ qua giá trị khi bằng 0 hoặc 1. Hoặc thêm một biến chứa giá trị enable, nguyên lý hoạt động khi enable thì cho phép dịch bộ nhớ, Nếu không thì không dịch bộ nhớ.



Hình 22: Bộ đếm và trích xuất tín hiệu enable

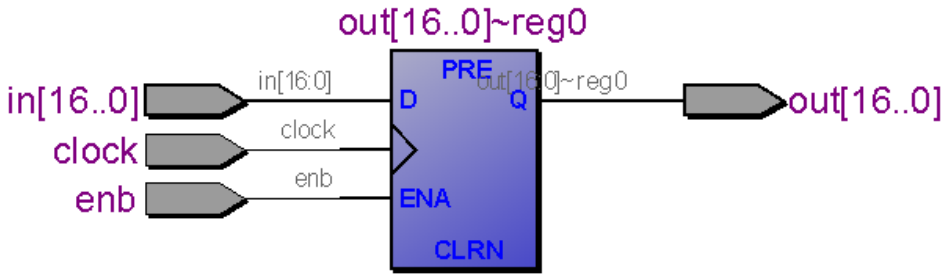
Tín hiệu enable đảo, tín hiệu này sẽ có giá trị là 0 khi biến đếm counter khác 0 hoặc 1 và bằng 1 khi counter bằng 0 hoặc 1. Biến đếm sẽ đếm từ 0 đến 9 và sẽ reset về 0 khi đặt bằng 10, Như vậy tín hiệu enable đã bỏ qua được hai giá trị rác khi counter bằng 0 hoặc 1.

Source code verilog bộ đếm và trích xuất tín hiệu enable:

```
reg [3:0] counter;
```

```
wire enable;
always @(posedge href) begin
        if (counter == 9) begin
                counter = 0;
                end
        else begin
                counter = counter + 1;
                end
end
assign enable = ~(~counter[3]&~counter[2]&~counter[1]);
```

Bộ nhớ đệm output feature map sẽ bao gồm 24 ô nhớ, mỗi ô nhớ cho 17 bit với bit MSB là bit dấu tương ứng với ngõ ra của core computing kernal. Các ô nhớ sẽ được đánh số từ 0 đến 23. Được tổ chức theo từng frame, mỗi frame gồm 8 ô nhớ. Được kết nỗi theo kiểu frame trước sẽ là đầu vào của frame sau nhằm tạo cơ chế dịch ô nhớ.



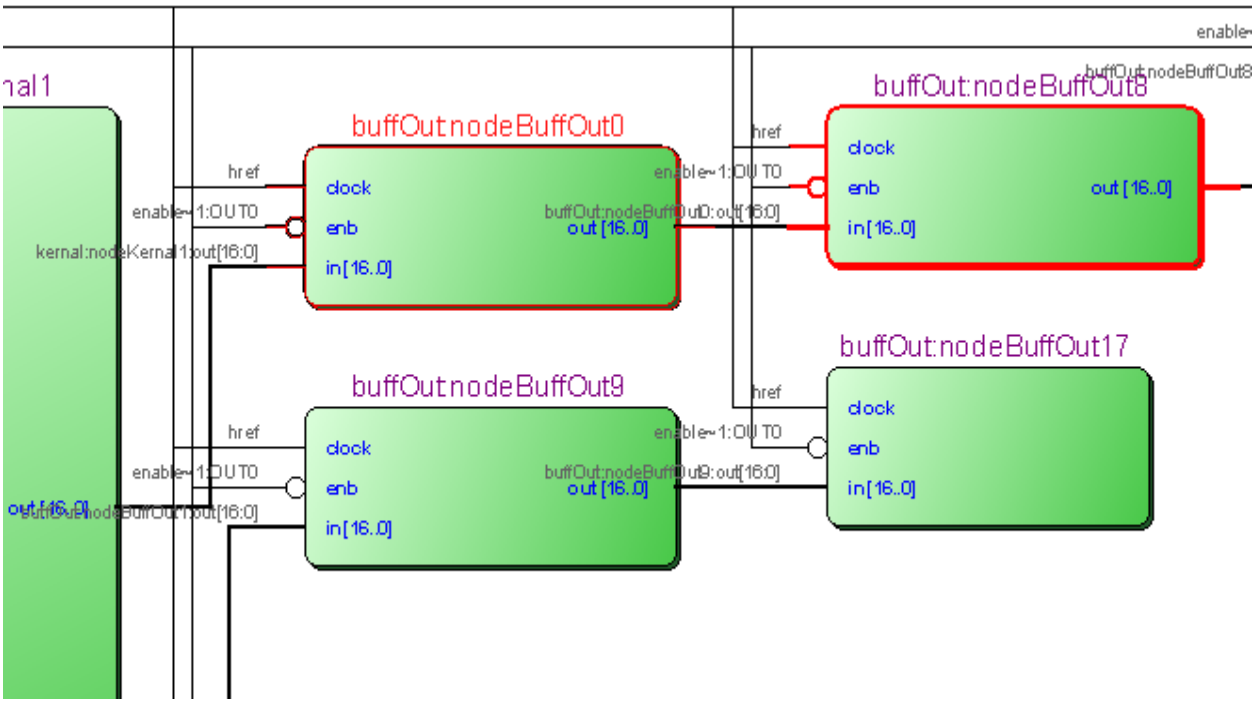
Hình 23: Cấu trúc bên trong của một ô nhớ output feature map

Hoạt động của ô nhớ feature map rất đơn giản khi có xung clock vào thì kiểm tra chân enb nếu mức cao thì sẽ gán đầu ra bằng với đầu vào. Nếu chân enb mức thấp thì sẽ giữ nguyên đầu ra ở bus out.

Source code verilog của một ô nhớ của feature map output:

```
module buffOut (in, out, clock, enb);
input clock;
input enb;
input [16:0] in;
output reg [16:0] out;
always @(posedge clock) begin
        if(enb == 1'b1) begin
                out <= in;
                end
        else begin
                out <= out;
                end
end
endmodule
```

Phần thiết kế bộ nhớ feature map, Vì lý do các ô nhớ được quartus phân bố rãi rác, không đồng nhất gây ra khó quan sát, Nên rất khó để trình bày toàn bộ ô nhớ nên trong bản báo cáo này chỉ trình bày sơ đồ cấu trúc, liên kết của một vài ô nhớ dễ quan sát, các ô nhớ còn lại được kết nối tương tự.



Hình 24: Một phần khiến trúc của bộ nhớ feature map

Vì kích thước của output feature map là 8x8 và ta cần tối thiểu 3 frame để trở thành đầu vào cho lớp tích chập tiếp theo mà mỗi frame có 8 ô nhớ vì vậy ta cần có 24 ô nhớ, các ô

nhớ được chia đều vào 3 frame, frame đầu tiên sẽ được kết nối với ngõ ra của core computing kernal, frame tiếp theo sẽ có đầu vào là frame trước đó để tạo thành cơ cấu trược dữ liệu phục vụ cho việc thực hiện convolution cho lớp tiếp theo.

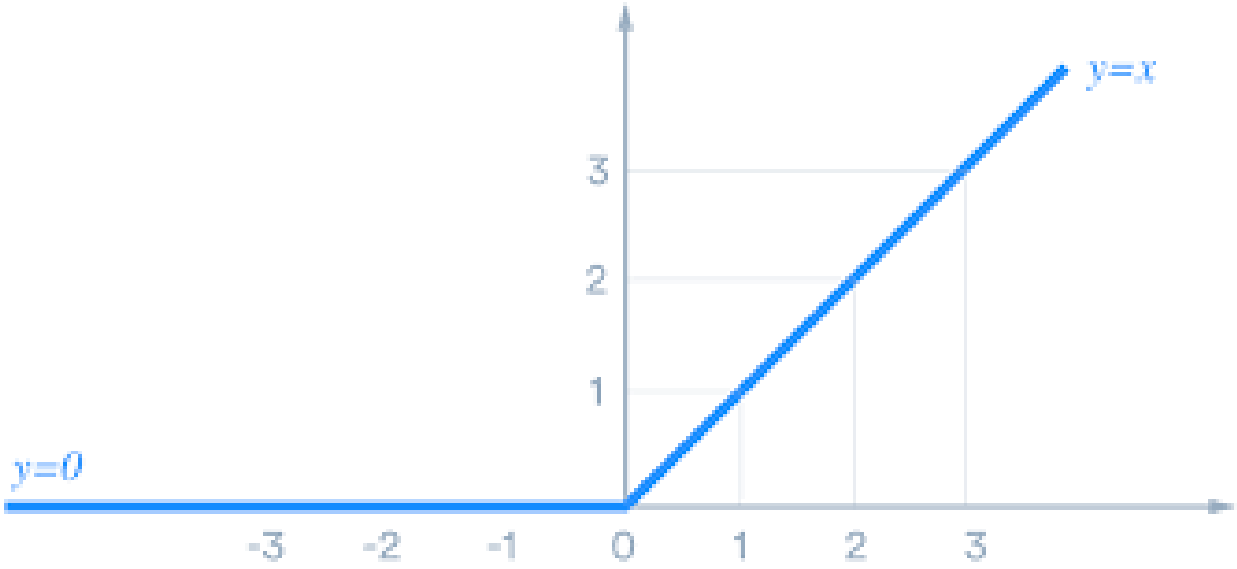
Source code verilog của bộ nhớ output feature map:

```
wire [16:0] bf [0:23];
buffOut nodeBuffOut0(conv1[0],bf[16], href, enable);
buffOut nodeBuffOut1(conv1[1],bf[17], href, enable);
buffOut nodeBuffOut2(conv1[2],bf[18], href, enable);
buffOut nodeBuffOut3(conv1[3],bf[19], href, enable);
buffOut nodeBuffOut4(conv1[4],bf[20], href, enable);
buffOut nodeBuffOut5(conv1[5],bf[21], href, enable);
buffOut nodeBuffOut6(conv1[6],bf[22], href, enable);
buffOut nodeBuffOut7(conv1[7],bf[23], href, enable);
buffOut nodeBuffOut8(bf[16],bf[8], href, enable);
buffOut nodeBuffOut9(bf[17],bf[9], href, enable);
buffOut nodeBuffOut10(bf[18],bf[10], href, enable);
buffOut nodeBuffOut11(bf[19],bf[11], href, enable);
buffOut nodeBuffOut12(bf[20],bf[12], href, enable);
buffOut nodeBuffOut13(bf[21],bf[13], href, enable);
buffOut nodeBuffOut14(bf[22],bf[14], href, enable);
buffOut nodeBuffOut15(bf[23],bf[15], href, enable);
buffOut nodeBuffOut16(bf[8],bf[0], href, enable);
buffOut nodeBuffOut17(bf[9],bf[1], href, enable);
buffOut nodeBuffOut18(bf[10],bf[2], href, enable);
buffOut nodeBuffOut19(bf[11],bf[3], href, enable);
buffOut nodeBuffOut20(bf[12],bf[4], href, enable);
buffOut nodeBuffOut21(bf[13],bf[5], href, enable);
buffOut nodeBuffOut22(bf[14],bf[6], href, enable);
buffOut nodeBuffOut23(bf[15],bf[7], href, enable);
```

1.4 Thiết kế activation function ReLU

Hàm activation function là một hàm rất quan trọng trong mô hình CNN, hàm này có chức năng khữ tuyến tính, giúp cho việc biểu diễn các bộ dữ liệu mang tính phi tuyển tốt hơn.

Đồ thị của hàm ReLU:

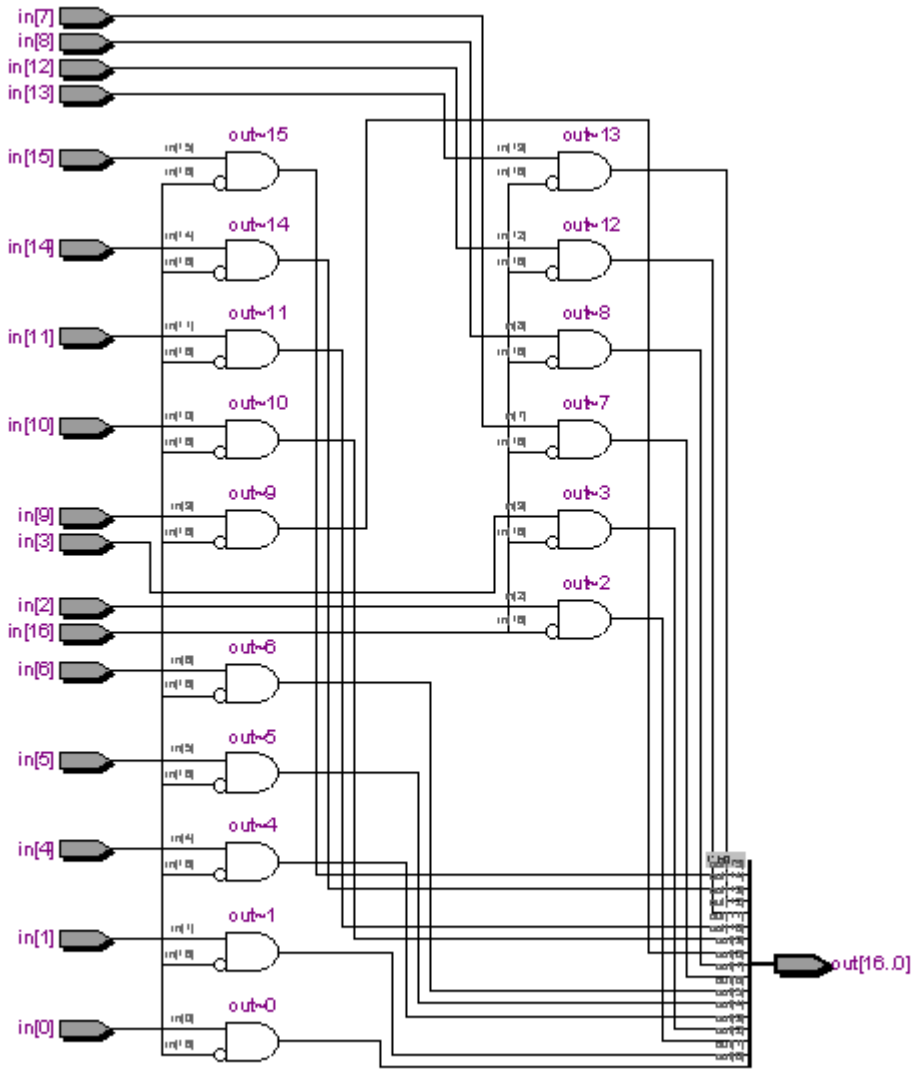


Hình 25: Đồ thị của hàm ReLU

Thông thường output feature map không cần phải qua hàm ReLU, tuy nhiên nếu output feature cần phải xuất ra màng hình qua VGA hoặc các phương thức tương tự. Khi muốn output feature map có thể xuất VGA hoặc các phương thức tương tự thì cần giới hạn các giá trị của feature map, Các giá trị feature map cần lớn hơn 0 và nhỏ hơn 255. Vì vậy ta có thể thực hiện giới hạn như sau:

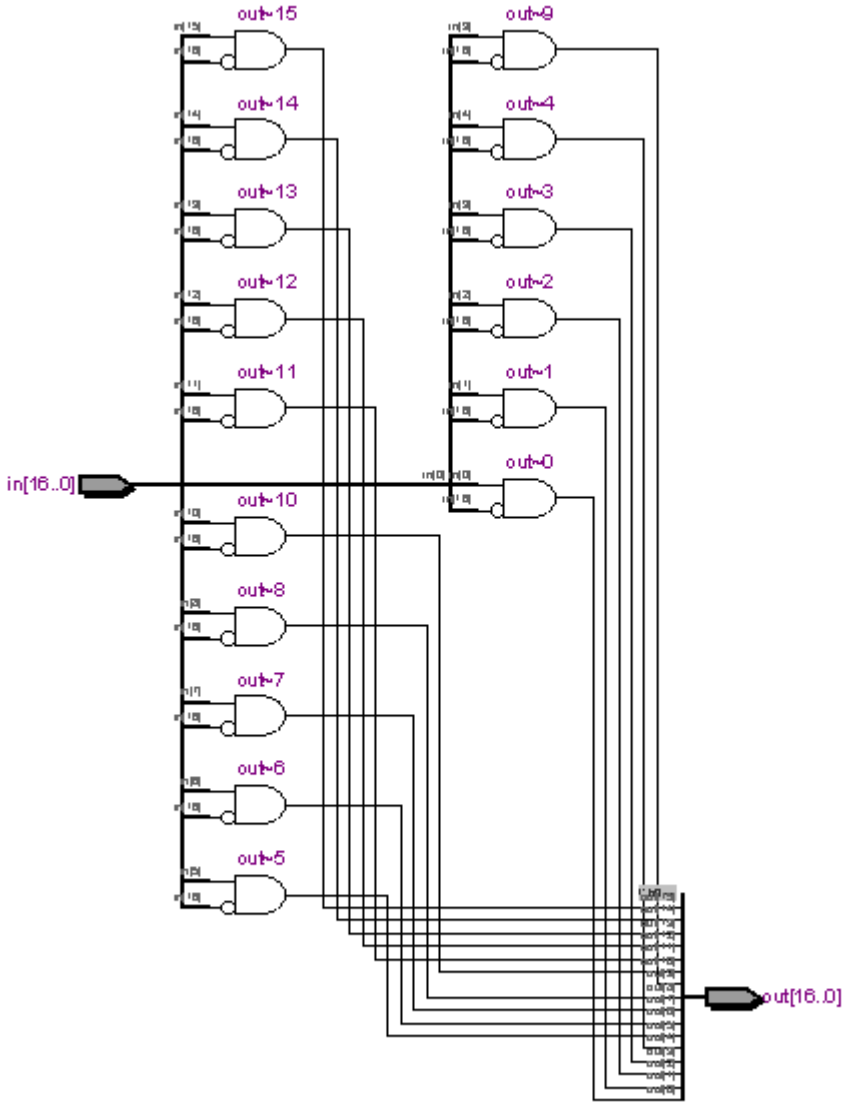
Nếu giá trị của feature map nhỏ hơn 0 tức feature map có MSB bit bằng 1. Vì vậy ta có thể thiết kế các bit từ 0 đến 15 and với not của MSB để chuyển các giá trị âm thành 0.

Như vậy ta có thể thực hiện hàm ReLU như sau:



Hình 26: Cấu trúc hàm ReLU dạng chi tiết.

Sơ đồ cấu trúc dạng rút gọn của hàm ReLU



Hình 27: Cấu trúc hàm ReLU dạng rút gọn

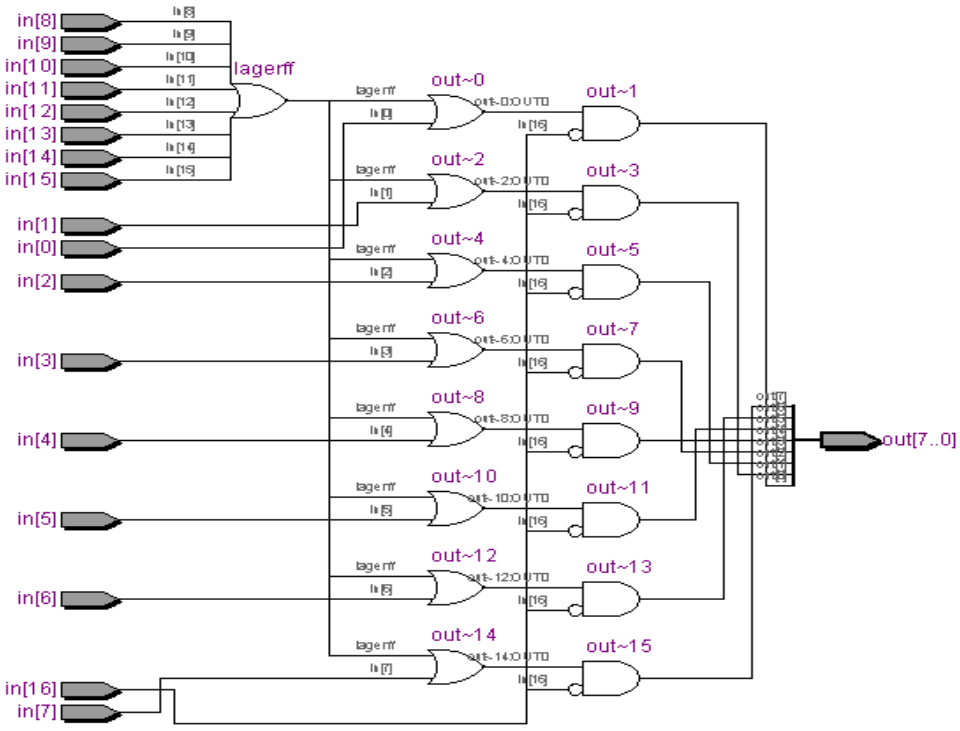
Source code verilog của ReLU function:

```
module relu(in, out);
input [16:0] in;
output [16:0] out;
assign out[0]=in[0]&(~in[16]);
assign out[1]=in[1]&(~in[16]);
assign out[2]=in[2]&(~in[16]);
assign out[3]=in[3]&(~in[16]);
assign out[4]=in[4]&(~in[16]);
```

```
assign out[5]=in[5]&(~in[16]);
assign out[6]=in[6]&(~in[16]);
assign out[7]=in[7]&(~in[16]);
assign out[8]=in[8]&(~in[16]);
assign out[9]=in[9]&(~in[16]);
assign out[10]=in[10]&(~in[16]);
assign out[11]=in[11]&(~in[16]);
assign out[12]=in[12]&(~in[16]);
assign out[13]=in[13]&(~in[16]);
assign out[14]=in[14]&(~in[16]);
assign out[15]=in[15]&(~in[16]);
assign out[16]=in[16]&(~in[16]);
endmodule
```

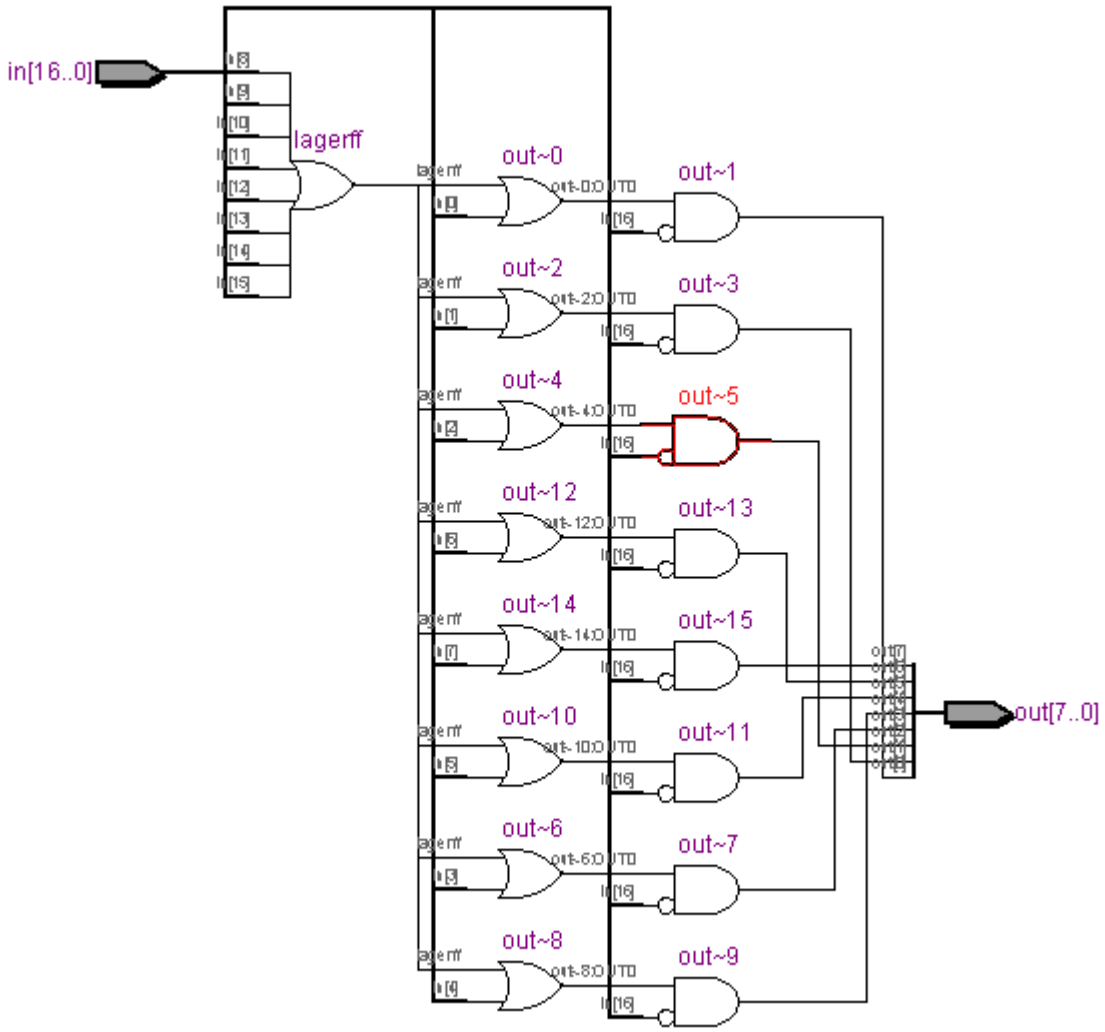
Nếu muốn xuất feature map theo phương thức VGA hoặc các phương thức tương tự thì cần loại bỏ số âm, đồng thời giới hạn của feature map nhỏ hơn hoặc bằng 255. Nếu feature map lớn hơn 255 thì chuyển giá trị này thành 255.

Ta có thể thực hiện giới hạn giá trị của feature map theo sơ đồ như sau:



Hình 28: Cấu trúc bên trong bộ giới hạn giá trị của feature map dạng đầy đủ

Dạng rút gọn của bộ giới hạn feature map:



Hình 29: Cấu trúc rút gọn của bộ giới hạn giá trị của feature map

Source code verilog của bộ giới hạn giá trị của output feature map:

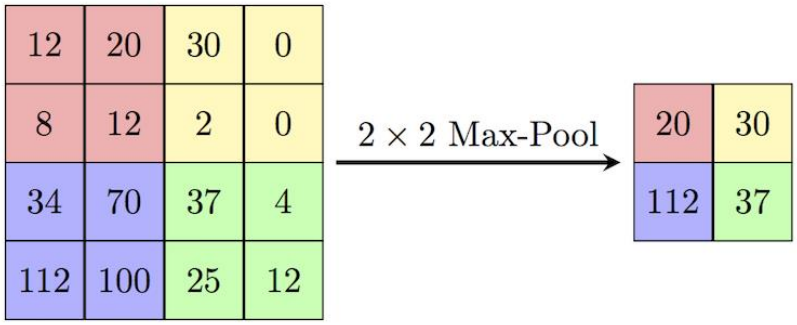
```
module gh(in, out);
input [16:0] in;
output [7:0] out;
wire lagerff;
assign lagerff = in[15]|in[14]|in[13]|in[12]|in[11]|in[10]|in[9]|in[8];
assign out[0] = (in[0]|lagerff)&(~in[16]);
assign out[1] = (in[1]|lagerff)&(~in[16]);
assign out[2] = (in[2]|lagerff)&(~in[16]);
assign out[3] = (in[3]|lagerff)&(~in[16]);
```

```
assign out[4] = (in[4]|lagerff)&(~in[16]);
assign out[5] = (in[5]|lagerff)&(~in[16]);
assign out[6] = (in[6]|lagerff)&(~in[16]);
assign out[7] = (in[7]|lagerff)&(~in[16]);
endmodule
```

1.5 Thực hiện thuật toán pooling.

Thuật toán pooling dùng để giảm bớt kích thước của dữ liệu sau khi đã qua lớp tích chập. Thuật toán pooling có nhiều loại khác nhau trong đó có hai loại được sử dụng phổ biến là max pooling và avg pooling.

Đối với max pooling sẽ cho đầu ra là giá trị lớn nhất trong các đầu vào.



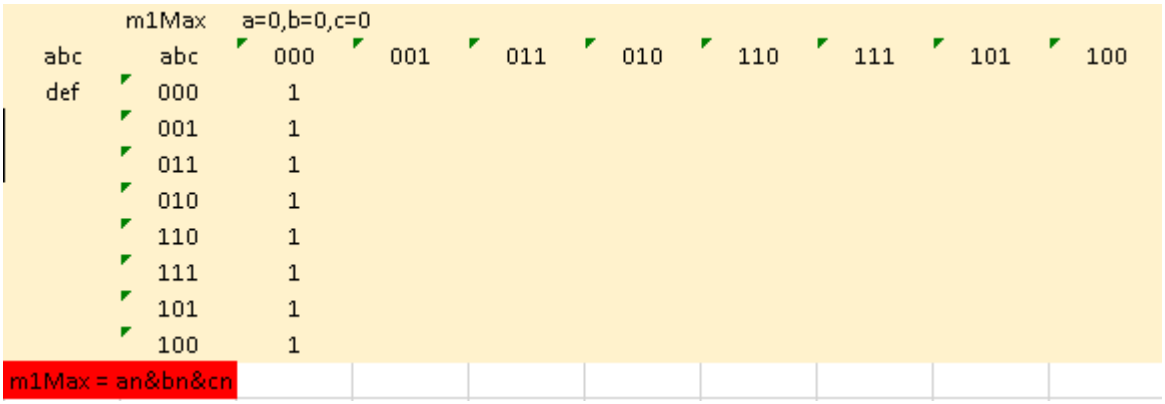
Hình 30: Minh họa thuật toán max-pooling

Để thực hiện được thuật toán trên ta có thể áp dụng một giải thuật như sau: ta tạo ra các tín hiệu tạm là bit MSB của phép trừ theo bảng sau:

m1 - m2	m1-m3	m1-m4	m2-m3	m2-m4	m3-m4
a	b	c	d	e	f

Hình 31: Bảng tín hiệu tạm của thuật toán pooling

Ta thực hiện bìa K cho tín hiệu lớn nhất là m1:



Hình 32: Bìa K cho tín hiệu max là m1

Tương tự như vậycho tín hiệu lớn nhất là m2:

	m2 max	a=1,d=0,e=0							
	abc	000	001	011	010	110	111	101	100
def	000					1	1	1	1
	001					1	1	1	1
	011								
	010								
	110								
	111								
	101								
	100								
m2Max = a&dn&en									

Hình 34: Bìa K cho tín hiệu max là m2

Tương tự như vậycho tín hiệu lớn nhất là m3:

	m3 max	b=1,d=1,f=0							
	abc	000	001	011	010	110	111	101	100
def	000								
	001								
	011								
	010								
	110			1	1	1	1		
	111								
	101								
	100			1	1	1	1		
m3Max = b&d&cn									

Hình 33: Bìa K cho tín hiệu max là m3

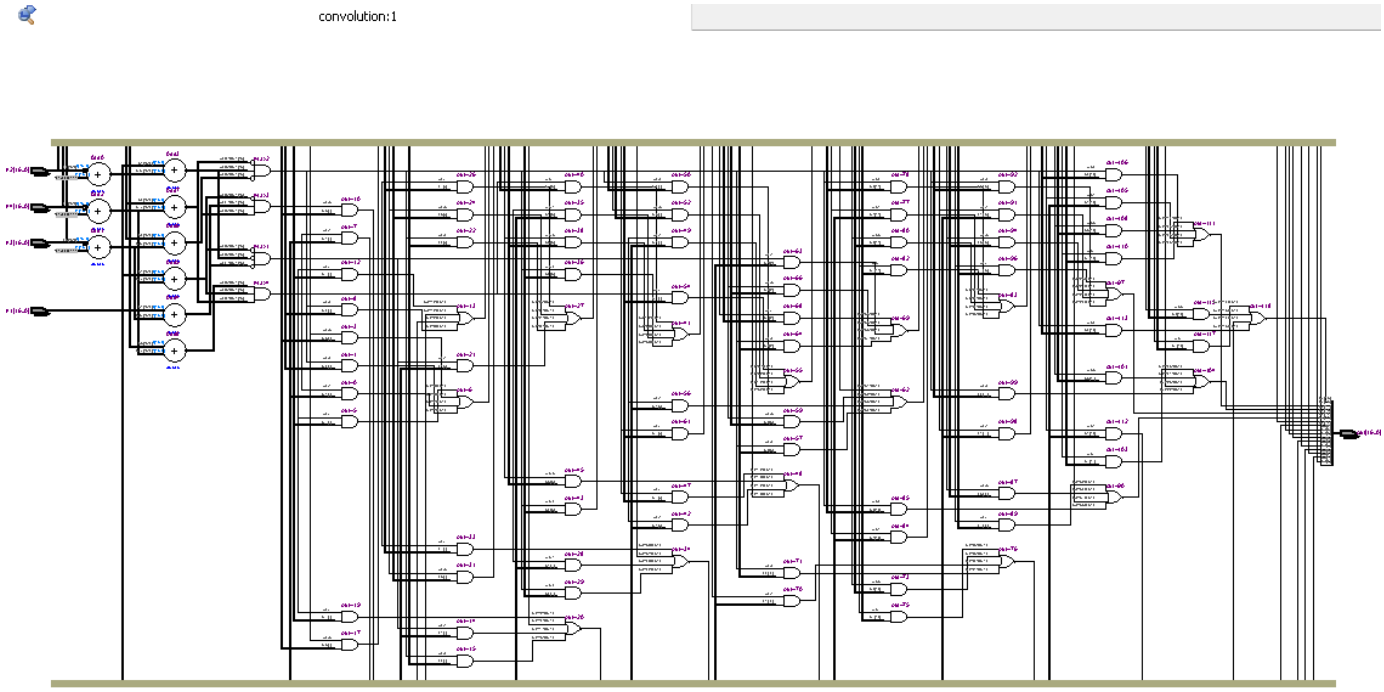
Tương tự như vậy cho tín hiệu lơn nhất là m4:

	m4 max	c=1,e=1,f=1							
	abc	000	001	011	010	110	111	101	100
def	000								
	001								
	011		1	1			1	1	
	010								
	110								
	111		1	1			1	1	
	101								
	100								
m4Max = c&e&f									

Hình 35: Bìa K cho tín hiệu lớn nhất là m4

Sau khi có được tín hiệu max1, max2, max3, max4. Ta chỉ cần lấy m1&max1 or m2&max2 or m3&max3 or m4&max4.

Ta có thể thiết kết bộ tính max-pooling như sau:



Hình 36: Cấu trúc bộ thuật toán max-pooling bốn ngõ vào 17 bit

Source code verilog của bộ tính thuật toán max-pooling 4 ngõ vào 17 bit.

```
module maxpool(in1,in2,in3,in4,out);
input [16:0] in1,in2,in3,in4;
output [16:0] out;
wire [16:0] in2nt, in3nt, in4nt;
wire [16:0] in2n, in3n, in4n;
assign in2nt[0]=~in2[0];
assign in2nt[1]=~in2[1];
assign in2nt[2]=~in2[2];
assign in2nt[3]=~in2[3];
assign in2nt[4]=~in2[4];
assign in2nt[5]=~in2[5];
assign in2nt[6]=~in2[6];
assign in2nt[7]=~in2[7];
assign in2nt[8]=~in2[8];
assign in2nt[9]=~in2[9];
assign in2nt[10]=~in2[10];
assign in2nt[11]=~in2[11];
assign in2nt[12]=~in2[12];
assign in2nt[13]=~in2[13];
assign in2nt[14]=~in2[14];
assign in2nt[15]=~in2[15];
assign in2nt[16]=~in2[16];
```

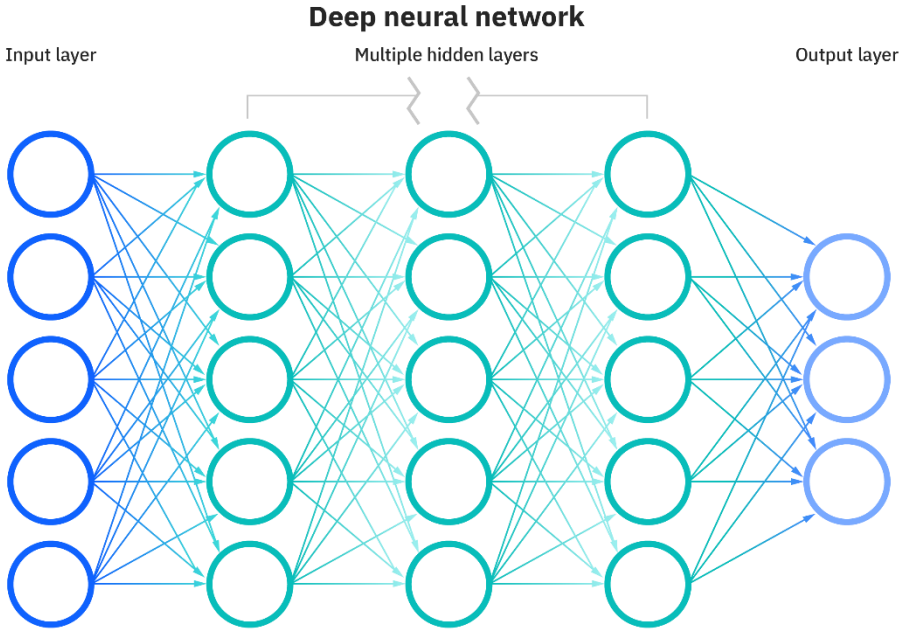
```
//=====in3
assign in3nt[0]=~in3[0];
assign in3nt[1]=~in3[1];
assign in3nt[2]=~in3[2];
assign in3nt[3]=~in3[3];
assign in3nt[4]=~in3[4];
assign in3nt[5]=~in3[5];
assign in3nt[6]=~in3[6];
assign in3nt[7]=~in3[7];
assign in3nt[8]=~in3[8];
assign in3nt[9]=~in3[9];
assign in3nt[10]=~in3[10];
assign in3nt[11]=~in3[11];
assign in3nt[12]=~in3[12];
assign in3nt[13]=~in3[13];
assign in3nt[14]=~in3[14];
assign in3nt[15]=~in3[15];
assign in3nt[16]=~in3[16];
//=======in4
assign in4nt[0]=~in4[0];
assign in4nt[1]=~in4[1];
assign in4nt[2]=~in4[2];
assign in4nt[3]=~in4[3];
assign in4nt[4]=~in4[4];
assign in4nt[5]=~in4[5];
assign in4nt[6]=~in4[6];
assign in4nt[7]=~in4[7];
assign in4nt[8]=~in4[8];
assign in4nt[9]=~in4[9];
assign in4nt[10]=~in4[10];
assign in4nt[11]=~in4[11];
assign in4nt[12]=~in4[12];
assign in4nt[13]=~in4[13];
assign in4nt[14]=~in4[14];
assign in4nt[15]=~in4[15];
assign in4nt[16]=~in4[16];
//==========in1n
assign in2n = in2nt + 17'b1;
assign in3n = in3nt + 17'b1;
assign in4n = in4nt + 17'b1;
//=======createSignal
wire [17:0] sub1, sub2, sub3, sub4, sub5, sub6;
assign sub1 = in1 + in2n;
```

```
        assign sub2 = in1 + in3n;
        assign sub3 = in1 + in4n;
        assign sub4 = in2 + in3n;
        assign sub5 = in2 + in4n;
        assign sub6 = in3 + in4n;
        wire a,b,c,d,e,f;
        assign a = sub1[16];
        assign b = sub2[16];
        assign c = sub3[16];
        assign d = sub4[16];
        assign e = sub5[16];
        assign f = sub6[16];
        wire max1, max2, max3, max4;
        assign max1 = (~a)&(~b)&(~c);
        assign max2 = (a)&(~d)&(~e);
        assign max3 = b&d&(~c);
        assign max4 = c&e&f;
        //=======createOut=
        assign out[0]=(in1[0]&max1)|(in2[0]&max2)|(in3[0]&max3)|(in4[0]&max4);
        assign out[1]=(in1[1]&max1)|(in2[1]&max2)|(in3[1]&max3)|(in4[1]&max4);
        assign out[2]=(in1[2]&max1)|(in2[2]&max2)|(in3[2]&max3)|(in4[2]&max4);
        assign out[3]=(in1[3]&max1)|(in2[3]&max2)|(in3[3]&max3)|(in4[3]&max4);
        assign out[4]=(in1[4]&max1)|(in2[4]&max2)|(in3[4]&max3)|(in4[4]&max4);
        assign out[5]=(in1[5]&max1)|(in2[5]&max2)|(in3[5]&max3)|(in4[5]&max4);
        assign out[6]=(in1[6]&max1)|(in2[6]&max2)|(in3[6]&max3)|(in4[6]&max4);
        assign out[7]=(in1[7]&max1)|(in2[7]&max2)|(in3[7]&max3)|(in4[7]&max4);
        assign out[8]=(in1[8]&max1)|(in2[8]&max2)|(in3[8]&max3)|(in4[8]&max4);
        assign out[9]=(in1[9]&max1)|(in2[9]&max2)|(in3[9]&max3)|(in4[9]&max4);
        assign
        out[10]=(in1[10]&max1)|(in2[10]&max2)|(in3[10]&max3)|(in4[10]&max4);
        assign
        out[11]=(in1[11]&max1)|(in2[11]&max2)|(in3[11]&max3)|(in4[11]&max4);
        assign
        out[12]=(in1[12]&max1)|(in2[12]&max2)|(in3[12]&max3)|(in4[12]&max4);
        assign
        out[13]=(in1[13]&max1)|(in2[13]&max2)|(in3[13]&max3)|(in4[13]&max4);
        assign
        out[14]=(in1[14]&max1)|(in2[14]&max2)|(in3[14]&max3)|(in4[14]&max4);
        assign
        out[15]=(in1[15]&max1)|(in2[15]&max2)|(in3[15]&max3)|(in4[15]&max4);
        assign
        out[16]=(in1[16]&max1)|(in2[16]&max2)|(in3[16]&max3)|(in4[16]&max4);
endmodule
```

Sơ đồ kiến trúc của bộ max-pooling có vẽ khá phức tạp, tuy nhiên sự phúc tạp này nằm ở số lượng bit có trong một số, Còn về nguyên lý hoạt động cơ bản thì vô cùng đơn giản. Bước đầu tiên sẽ thực hiện tìm các số đảo dấu của số thứ 2, 3, 4. Tiếp theo cho số thứ 1 – số thứ 2, số thứ 1 – số thứ 3, số thứ 1 – số thứ 4, số thứ 2 – số thứ 3, số thứ 2 – số thứ 4, số thứ 3 – số thứ 4. Sau đó lần lượt gán bit MSB của các kết quả cho các tín hiệu lần lượt là a, b, c, d, e, f. Dựa vào các tín hiệu này và các bảng bìa K đã trình bày phía trên để xuất ra các tin hiệu max1 (Bằng 1 khi max là số thứ 1, 0 nếu số thứ 1 không phải là số lớn nhất), tượng tự vậy với max2, max3, max4. Từ các giá trị max1, max2, max3, max4. Ta có thể thực hiện kiến trúc cổng logic để xuất đầu ra đơn giản như sau: out = (inp1&max1)|(inp2&max2)|(inp3&max3)|(inp4&max4).

2. Thực hiện phần fully connected.

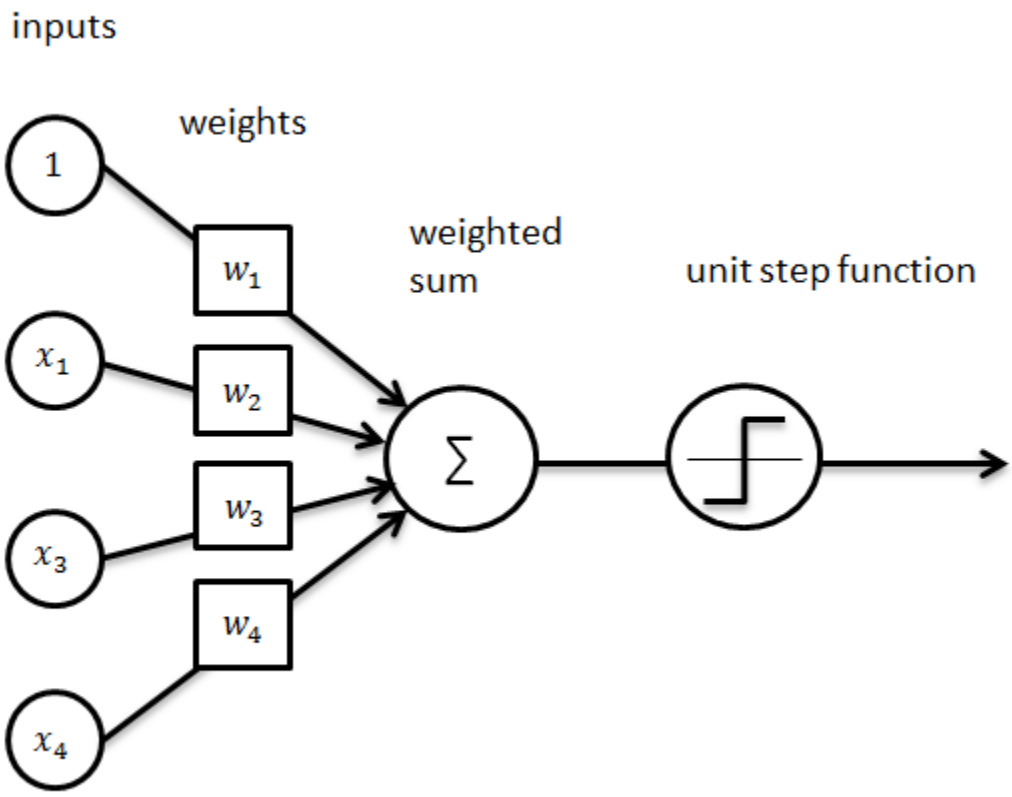
Khác với convolution layer dữ liệu đầu vào sẽ được tính toán theo từng vùng, các vùng này chống lần lên nhau, nhưng tại mỗi đơn vị tính toán chỉ thức hiện tính toán trên một vùng nhất định mà không cần tính toán trên toàn bộ dữ liệu đầu vào. Đối với lớp fully connected thì lớp neural network sau sẽ kết nối với toàn bộ dữ liệu đầu vào hay lớp neural trước đó.



Hình 37: Hoạt động của fully connected

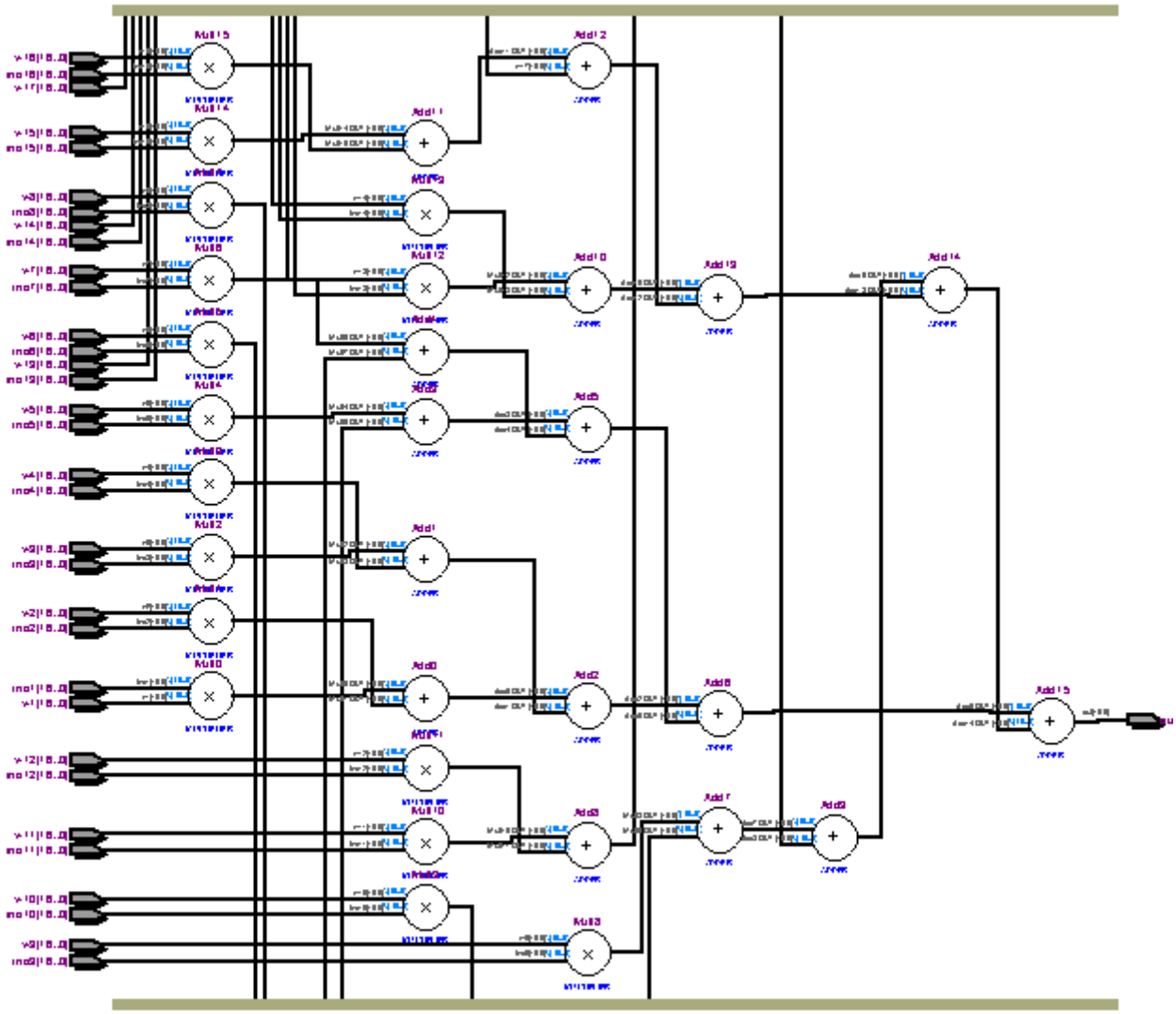
Hoạt động của một node neural network rất đơn giản, 1 node neural network ở layer này sẽ nối với tấc cả các output của layer trước đó để thực hiện phép toán tích chập. Tiếp theo ta được kết qua sẽ cho đi qua hàm activation, và dữ liệu sẽ được đẩy ra đầu ra của

node neural network, Các node neural của một lớp sẽ tạo thành output của layer và sẽ tiếp tục như vậy với layer tiếp theo.



Hình 38: Minh họa hoạt động của một node neural

Từ nguyên lý hoạt động của một node neural như trên ta có thể xây dựng mẫu một node neural gồm 16 ngõ vào, 17 trọng số, mỗi số gồm 17bit như sau:



Hình 39: Cấu trúc bên trong của một node neural

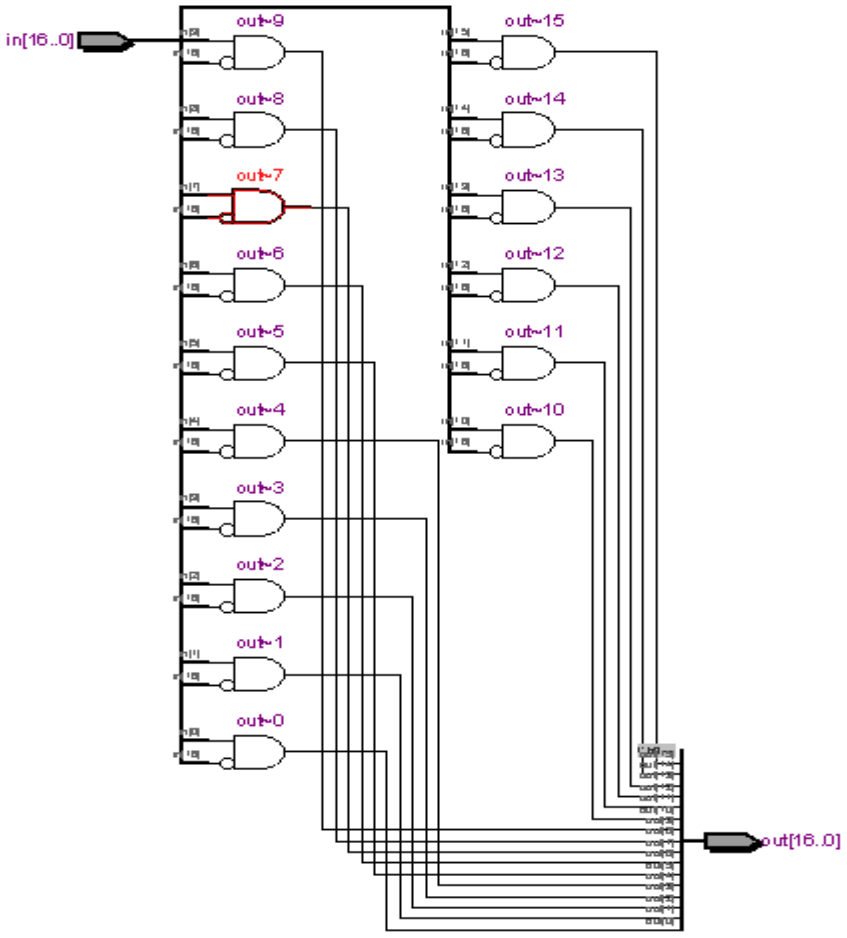
Ta có thể thấy rằng node neural gồm 16 ngõ vào, 17 trong số trong đó có một trọng số là hệ số bias. Kiến trúc sử dụng 16 bộ nhân được sắp xếp 1 tầng giúp giảm thời gian delay. Đồng thời các bộ cộng được sắp xếp tối đa 5 tầng vì vậy cũng cố gắng tối ưu về mặt delay.

Source code verilog cho node neural gồm 16 ngõ vào, 17 trọng số, mỗi số 17 bit với MSB là bit dấu như sau:

```
module
neural17b(ino1,ino2,ino3,ino4,ino5,ino6,ino7,ino8,ino9,ino10,ino11,ino12,ino13,ino14
,ino15,ino16,
w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,w11,w12,w13,w14,w15,w16,w17, out);
```

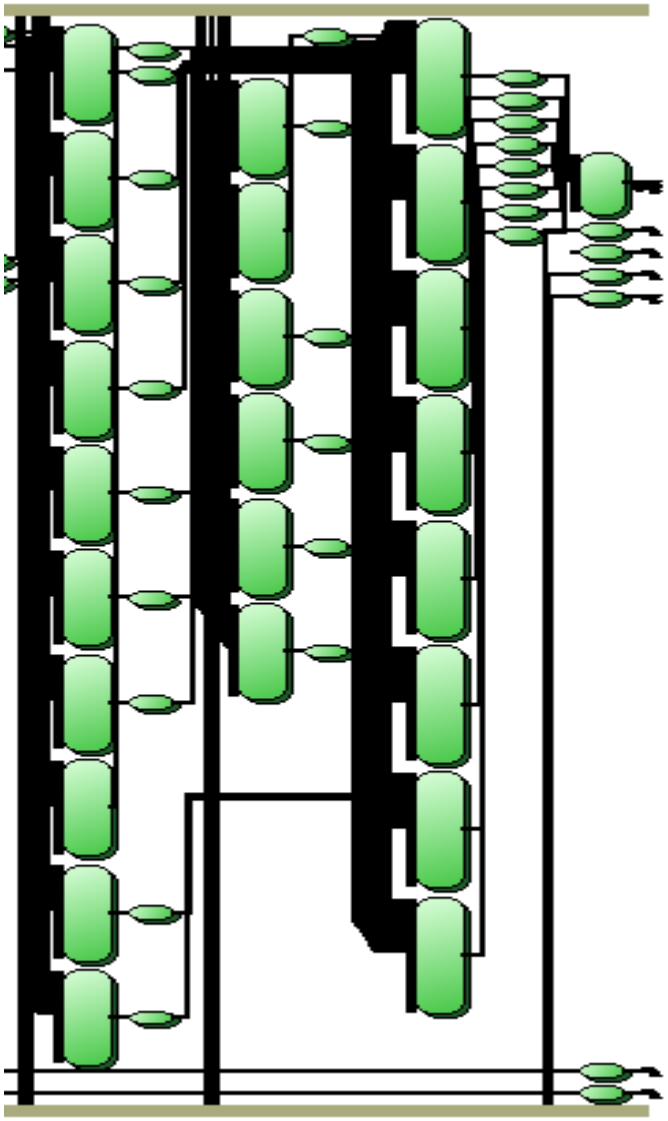
```
input [16:0] ino1,ino2,ino3,ino4,ino5,ino6,ino7,ino8,ino9, ino10,ino11,ino12,ino13, ino14, ino15,ino16;
input [16:0] w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,w11,w12,w13,w14,w15,w16,w17;
output [16:0] out;
assign out = (((ino1*w1 + w2*ino2) + (w3*ino3+ w4*ino4)) + ((w5*ino5+ w6*ino6) + (w7*ino7 + w8*ino8))) +
(((w9*ino9 + w10*ino10) + (w11*ino11 + w12*ino12)) + ((w13*ino13 + w14*ino14) + (w15*ino15 + w16*ino16 + w17)));
endmodule
```

Đối với hàm activation thường dùng là hàm ReLU cũng sẽ được thiết kế tương tự với hàm ReLU trong convolution layer như đã tìm hiểu phía trên. Tuy nhiên không cần phải giới hạn giá trị của kết quả đầu ra của node. Ta có sơ đồ của hàm activation ReLU như sau:



Hình 40: Activation function ReLU trong fully connected

Sau khi đã có các node và activation funciton, giờ ta chỉ cần kết nối lại với nhau để tạo thành mạng fully connected là hoàn thành.



Hình 41: Mạng fully connected với 2 lớp

Với sơ đồ kiến trúc phía trên có thể sẽ rất dễ nhầm lần rằng có 3 lớp tuy nhiên do sự sắp xếp của quartus đã sắp xếp thành 3 lớp, tuy nhiên kiến trúc có 2 lớp, lớp đầu tiên gồm 16 node, lớp thứ 2 gồm 8 node, Các lớp này được liên kết với nhau theo dạng fully connected. Theo sau mỗi lớp là một hàm activation ReLU.

Source code verilog của mạng fully connected phía trên như sau:

```
//========== ============Layer1================================


neural
layer0_0(inp1,inp2,inp3,inp4,inp5,inp6,inp7,inp8,inp9,inp10,inp11,inp12,inp13,-
17'd75,17'd213,
```

```
17'd174,-17'd70,17'd68,-17'd248,-17'd38,17'd35,17'd150,17'd57,-
17'd80,17'd231,17'd60,-17'd225,oLayer1_0);
neural
layer0_1(inp1,inp2,inp3,inp4,inp5,inp6,inp7,inp8,inp9,inp10,inp11,inp12,inp13,17'd10
7,17'd134,
17'd48,-17'd68,17'd108,-17'd53,17'd53,-17'd17,17'd160,17'd42,17'd27,17'd254,-
17'd60,-17'd66,oLayer1_1);
neural
layer0_2(inp1,inp2,inp3,inp4,inp5,inp6,inp7,inp8,inp9,inp10,inp11,inp12,inp13,17'd79,
17'd94,
17'd185,17'd57,-17'd10,17'd7,17'd75,17'd68,17'd162,-17'd39,17'd37,17'd211,17'd30,-
17'd75,oLayer1_2);
neural
layer0_3(inp1,inp2,inp3,inp4,inp5,inp6,inp7,inp8,inp9,inp10,inp11,inp12,inp13,17'd12
1,-17'd256,
-17'd48,17'd42,-17'd24,17'd30,-17'd104,17'd29,-17'd62,-17'd98,17'd73,-17'd138,-
17'd131,17'd92,oLayer1_3);
neural
layer0_4(inp1,inp2,inp3,inp4,inp5,inp6,inp7,inp8,inp9,inp10,inp11,inp12,inp13,-
17'd71,17'd80,
17'd109,17'd14,-17'd39,-17'd45,17'd95,-17'd53,17'd23,17'd0,17'd79,-17'd80,-
17'd23,17'd0,oLayer1_4);
neural
layer0_5(inp1,inp2,inp3,inp4,inp5,inp6,inp7,inp8,inp9,inp10,inp11,inp12,inp13,17'd11
2,17'd77,
17'd163,17'd120,17'd9,-17'd47,17'd103,17'd78,17'd41,-17'd60,-17'd48,17'd157,-
17'd35,-17'd48,oLayer1_5);
neural
layer0_6(inp1,inp2,inp3,inp4,inp5,inp6,inp7,inp8,inp9,inp10,inp11,inp12,inp13,-
17'd29,-17'd59,
17'd33,17'd80,-17'd107,-17'd71,17'd44,-17'd91,-17'd108,-17'd70,-17'd3,-17'd89,-
17'd100,17'd0,oLayer1_6);
neural
layer0_7(inp1,inp2,inp3,inp4,inp5,inp6,inp7,inp8,inp9,inp10,inp11,inp12,inp13,-
17'd94,17'd18,
-17'd108,-17'd87,-17'd1,-17'd30,17'd71,17'd34,-17'd20,17'd66,-17'd75,-
17'd86,17'd96,17'd0,oLayer1_7);
neural
layer0_8(inp1,inp2,inp3,inp4,inp5,inp6,inp7,inp8,inp9,inp10,inp11,inp12,inp13,-
17'd37,17'd63,
```

```
17'd44,-17'd81,-17'd84,-17'd64,-
17'd81,17'd109,17'd20,17'd4,17'd90,17'd16,17'd94,17'd0,oLayer1_8);
neural
layer0_9(inp1,inp2,inp3,inp4,inp5,inp6,inp7,inp8,inp9,inp10,inp11,inp12,inp13,17'd10
8,-17'd99,
17'd102,-17'd81,-17'd83,17'd6,17'd40,17'd33,-17'd63,-17'd107,17'd26,17'd103,-
17'd9,17'd0,oLayer1_9);
neural
layer0_10(inp1,inp2,inp3,inp4,inp5,inp6,inp7,inp8,inp9,inp10,inp11,inp12,inp13,17'd1
2,-17'd101,
-17'd77,17'd24,-17'd111,-17'd19,-17'd19,17'd29,-17'd76,-17'd77,17'd90,-
17'd14,17'd20,17'd0,oLayer1_10);
neural
layer0_11(inp1,inp2,inp3,inp4,inp5,inp6,inp7,inp8,inp9,inp10,inp11,inp12,inp13,-
17'd20,-17'd172,
-17'd175,-17'd74,17'd86,17'd0,17'd36,-17'd6,-17'd207,17'd44,-17'd68,-17'd296,-
17'd56,17'd114,oLayer1_11);
neural
layer0_12(inp1,inp2,inp3,inp4,inp5,inp6,inp7,inp8,inp9,inp10,inp11,inp12,inp13,-
17'd12,17'd12,
17'd28,17'd9,-17'd100,17'd14,-17'd31,-17'd92,17'd30,17'd55,-
17'd32,17'd64,17'd31,17'd0,oLayer1_12);
neural
layer0_13(inp1,inp2,inp3,inp4,inp5,inp6,inp7,inp8,inp9,inp10,inp11,inp12,inp13,17'd7
6,-17'd49,
17'd85,-17'd88,-17'd68,17'd22,-17'd114,-17'd15,-17'd99,-
17'd91,17'd106,17'd97,17'd89,17'd0,oLayer1_13);
neural
layer0_14(inp1,inp2,inp3,inp4,inp5,inp6,inp7,inp8,inp9,inp10,inp11,inp12,inp13,17'd4
4,-17'd239,
-17'd29,-17'd43,17'd74,17'd136,-17'd41,-17'd14,-17'd21,17'd61,17'd14,-17'd93,-
17'd135,17'd65,oLayer1_14);
neural
layer0_15(inp1,inp2,inp3,inp4,inp5,inp6,inp7,inp8,inp9,inp10,inp11,inp12,inp13,-
17'd65,-17'd105,
-17'd102,17'd25,-17'd35,17'd37,-17'd44,17'd109,-17'd208,17'd75,-17'd19,-17'd227,-
17'd175,17'd70,oLayer1_15);
//=========================================EndLayer1==========
=================================
```

```
//============================Activation
layer1=================================================
reluf noderelu0(oLayer1_0,outL1_1);
reluf noderelu1(oLayer1_1,outL1_2);
reluf noderelu2(oLayer1_2,outL1_3);
reluf noderelu3(oLayer1_3,outL1_4);
reluf noderelu4(oLayer1_4,outL1_5);
reluf noderelu5(oLayer1_5,outL1_6);
reluf noderelu6(oLayer1_6,outL1_7);
reluf noderelu7(oLayer1_7,outL1_8);
reluf noderelu8(oLayer1_8,outL1_9);
reluf noderelu9(oLayer1_9,outL1_10);
reluf noderelu10(oLayer1_10,outL1_11);
reluf noderelu11(oLayer1_11,outL1_12);
reluf noderelu12(oLayer1_12,outL1_13);
reluf noderelu13(oLayer1_13,outL1_14);
reluf noderelu14(oLayer1_14,outL1_15);
reluf noderelu15(oLayer1_15,outL1_16);
//=====================================End_Activation_Relu======
==================================
wire [16:0]
oLayer2_0,oLayer2_1,oLayer2_2,oLayer2_3,oLayer2_4,oLayer2_5,oLayer2_6,oLayer
2_7;
wire [16:0] outL2_1,outL2_2,outL2_3,outL2_4,outL2_5,outL2_6,outL2_7,outL2_8;
//=======================================Layer2==============
==================================
neural17b
layer1_0(outL1_1,outL1_2,outL1_3,outL1_4,outL1_5,outL1_6,outL1_7,outL1_8,outL
1_9,outL1_10,outL1_11,outL1_12,outL1_13,
outL1_14,outL1_15,outL1_16,17'd82,17'd50,-17'd56,-17'd16,17'd46,17'd71,-17'd4,-
17'd112,-17'd9,17'd122,17'd125,17'd115,-17'd45,
17'd65,17'd18,-17'd9,17'd56,oLayer2_0);
neural17b
layer1_1(outL1_1,outL1_2,outL1_3,outL1_4,outL1_5,outL1_6,outL1_7,outL1_8,outL
1_9,outL1_10,outL1_11,
```

```
outL1_12,outL1_13,outL1_14,outL1_15,outL1_16,-17'd36,-17'd11,-
17'd4,17'd73,17'd14,17'd89,17'd86,-17'd76,17'd118,
17'd122,-17'd100,17'd41,17'd39,17'd76,-17'd14,-17'd135,-17'd54,oLayer2_1);
neural17b
layer1_2(outL1_1,outL1_2,outL1_3,outL1_4,outL1_5,outL1_6,outL1_7,outL1_8,outL
1_9,outL1_10,outL1_11,
outL1_12,outL1_13,outL1_14,outL1_15,outL1_16,17'd22,-17'd35,-17'd7,17'd68,-
17'd46,-17'd8,-17'd8,17'd106,-17'd42,
17'd65,-17'd96,17'd123,17'd6,-17'd9,17'd84,-17'd20,17'd59,oLayer2_2);
neural17b
layer1_3(outL1_1,outL1_2,outL1_3,outL1_4,outL1_5,outL1_6,outL1_7,outL1_8,outL
1_9,outL1_10,outL1_11,
outL1_12,outL1_13,outL1_14,outL1_15,outL1_16,-17'd63,-17'd26,-
17'd11,17'd131,17'd109,17'd71,17'd70,17'd8,-17'd117,
-17'd12,-17'd120,17'd50,17'd36,-17'd7,-17'd64,-17'd25,17'd59,oLayer2_3);
neural17b
layer1_4(outL1_1,outL1_2,outL1_3,outL1_4,outL1_5,outL1_6,outL1_7,outL1_8,outL
1_9,outL1_10,outL1_11,
outL1_12,outL1_13,outL1_14,outL1_15,outL1_16,-17'd138,17'd35,17'd102,17'd15,-
17'd34,17'd41,-17'd30,-17'd66,-17'd21,
-17'd46,-17'd87,17'd123,17'd25,-17'd31,-17'd86,-17'd113,-17'd41,oLayer2_4);
neural17b
layer1_5(outL1_1,outL1_2,outL1_3,outL1_4,outL1_5,outL1_6,outL1_7,outL1_8,outL
1_9,outL1_10,outL1_11,
outL1_12,outL1_13,outL1_14,outL1_15,outL1_16,-17'd92,-17'd111,-17'd126,-
17'd104,17'd52,17'd123,17'd15,-17'd70,
17'd69,-17'd26,-17'd42,-17'd103,17'd114,17'd7,-17'd18,-17'd108,17'd0,oLayer2_5);
neural17b
layer1_6(outL1_1,outL1_2,outL1_3,outL1_4,outL1_5,outL1_6,outL1_7,outL1_8,outL
1_9,outL1_10,outL1_11,
outL1_12,outL1_13,outL1_14,outL1_15,outL1_16,17'd103,-
17'd21,17'd37,17'd50,17'd26,17'd60,-17'd56,17'd71,17'd98,
17'd18,17'd10,17'd124,17'd77,-17'd38,17'd58,-17'd62,-17'd55,oLayer2_6);
neural17b
layer1_7(outL1_1,outL1_2,outL1_3,outL1_4,outL1_5,outL1_6,outL1_7,outL1_8,outL
1_9,outL1_10,outL1_11,
outL1_12,outL1_13,outL1_14,outL1_15,outL1_16,-17'd114,17'd123,-17'd51,17'd32,-
17'd43,-17'd35,17'd8,17'd52,-17'd68,
-17'd52,17'd68,-17'd123,-17'd22,17'd111,-17'd121,17'd107,17'd0,oLayer2_7);
```

```
//=====================================EndLayer2==========
=================================
//============================Activation
layer1=================================================
reluf nodereluL2_0(oLayer2_0,outL2_1);
reluf nodereluL2_1(oLayer2_1,outL2_2);
reluf nodereluL2_2(oLayer2_2,outL2_3);
reluf nodereluL2_3(oLayer2_3,outL2_4);
reluf nodereluL2_4(oLayer2_4,outL2_5);
reluf nodereluL2_5(oLayer2_5,outL2_6);
reluf nodereluL2_6(oLayer2_6,outL2_7);
reluf nodereluL2_7(oLayer2_7,outL2_8);
//=====================================End_Activation_Relu======
=================================
wire [16:0] tempout;
//=====================================Layer3=================
=================================
finalNeural
finalNode(outL2_1,outL2_2,outL2_3,outL2_4,outL2_5,outL2_6,outL2_7,outL2_8,-
17'd15,17'd82,-17'd207,
-17'd146,17'd10,-17'd111,17'd82,-17'd68,-17'd56,tempout);
//=====================================EndLayer3==============
=================================

//============================Activation
sigmoid=================================================
assign LEDR = tempout;
assign LEDG[0] = (~tempout[16]);
assign LEDG[1] = (~tempout[16]);
assign LEDG[2] = (~tempout[16]);
assign LEDG[3] = (~tempout[16]);

assign LEDG[7] = tempout[16];
assign LEDG[6] = tempout[16];
assign LEDG[5] = tempout[16];
assign LEDG[4] = tempout[16];
```

```
//=====================================endActivation sigmoid=====================================
```

PHẦN 4: HƯỚNG PHÁT TRIỂN CỦA ĐỀ TÀI

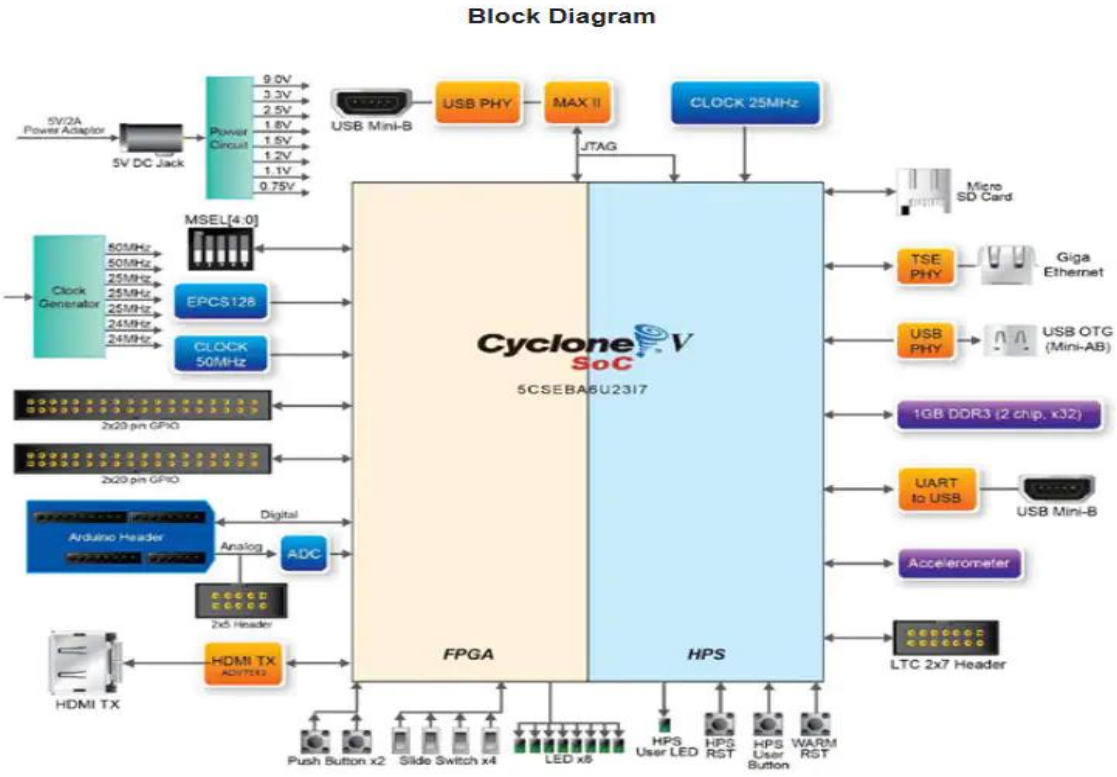
1. Những khó khăn của đề tài nghiêm cứu.

Như đã trình bày phía trên ta thấy việc nhận dữ liệu từ camera, đưa dữ liệu vào buffer được thiết kế rất khó khăn khi thực hiện bằng FPGA đơn thuần, sẽ không phù hợp để thực hiện những mô hình CNN lớn hớn, có kích thước ảnh đầu vào lớn hơn, nhiều lớp convolution, nhiều lớp pooling, nhiều lớp fully connected. Khi mô hình CNN trở nên lớn hơn, việc thiết kế chỉ dựa vào FPGA đơn thuần rất khó để thực hiện và dần trở nên khó thực hiện cũng như khó timing xung clock hay tín toán dalay buffer cho hợp lý. Đồng thời việc xuất kết quả qua VGA cũng trở nên khó khăn hơn.

Ngoài ra khi thiết kế FPGA đơn thuần rất khó để giao tiếp với các ngoại vi như màng hình LCD, UART, I2C,... Vì vậy việc thiết kế trở nên khó khăn và dễ xãy ra sai sót.

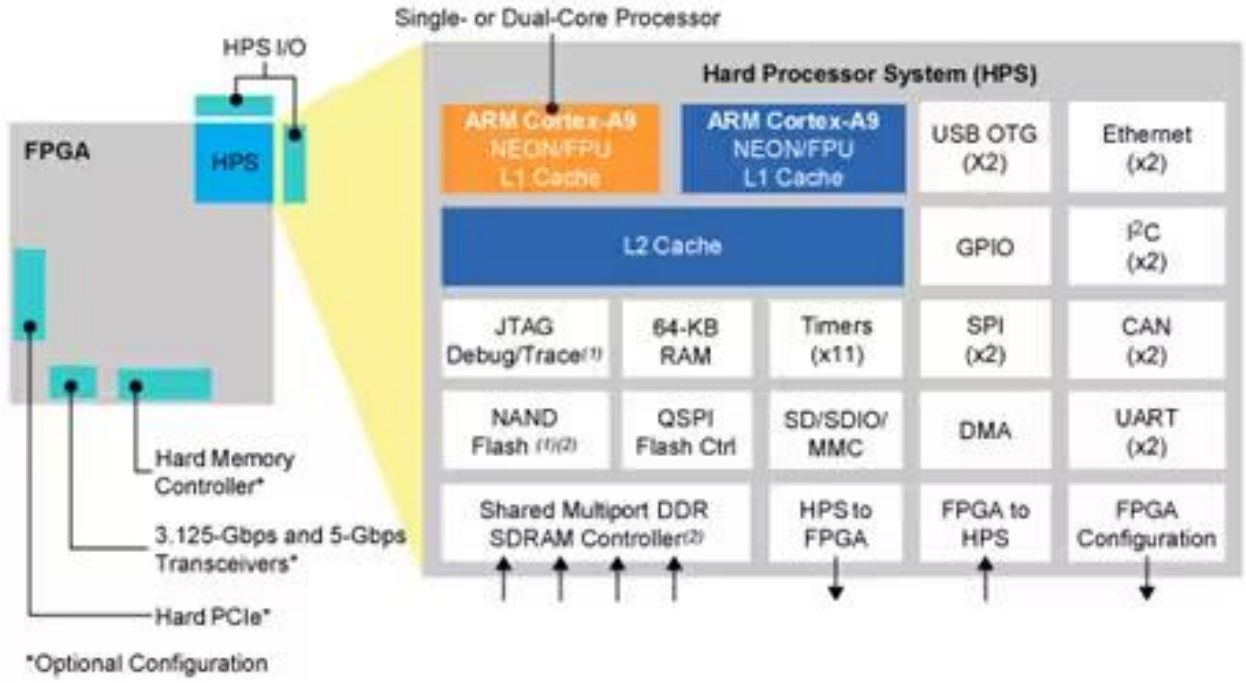
2. Thực hiện mô hình CNN trên SoC (system on chip)

Thuật ngữ SoC là viết tắt của System on a Chip. Chip này có tên như vậy bởi vì nó bao gồm nhiều thành phần tính toán thiết yếu, tất cả đều được nén vào một con chip. SoC chủ yếu được sử dụng cho các thiết bị di động vì kích thước nhỏ và sử dụng ít điện năng.



Hình 42: System on chip của dòng Cyclone V

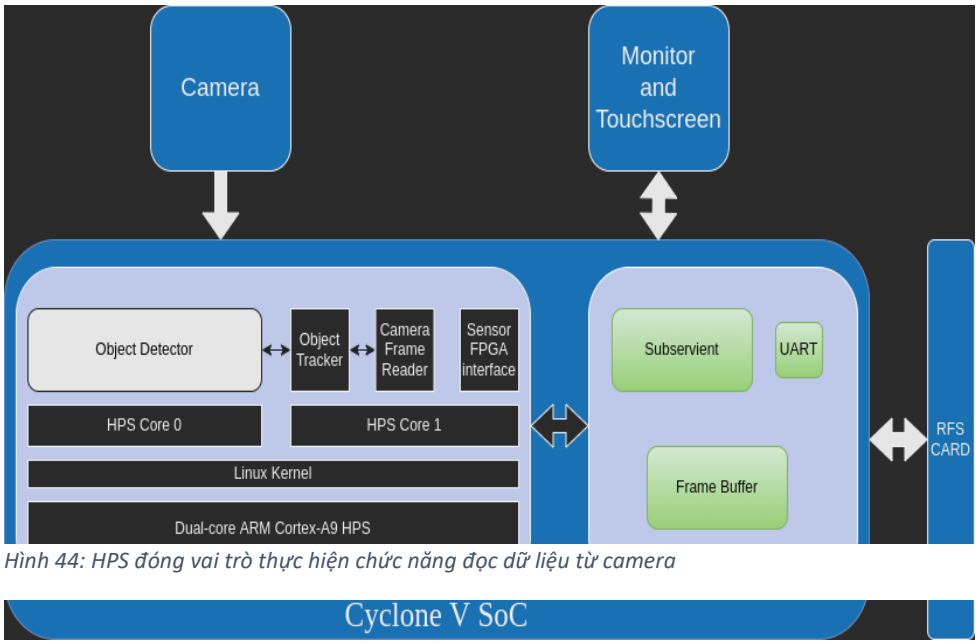
Như ta thấy dòng Cyclone V ngoại trừ phần FPGA còn có phần HPS là viết tắt của từ Hard Processor System, trên Cyclone V là ARM Cortex A9 Dual-Core processer.



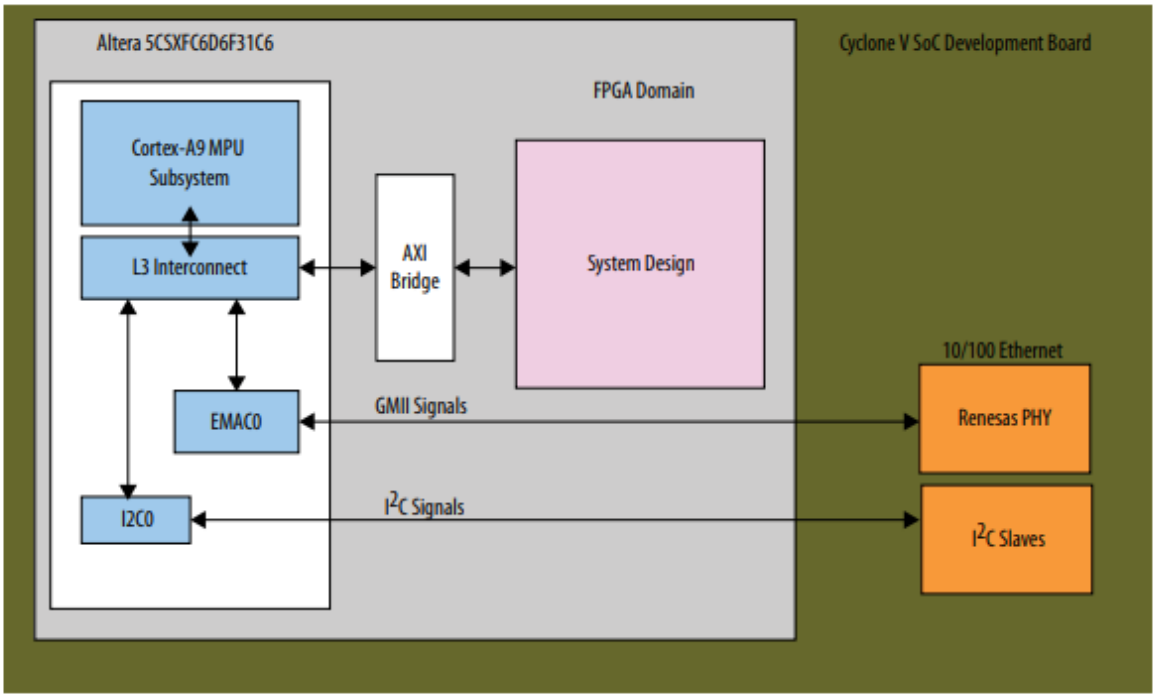
Hình 43: Kiến trúc HPS và các kết nối với FPGA của Cyclone V

HPS có thể giao tiếp với FPGA thông qua block HPS to FPGA, ngược lại thì FPGA có thể giao tiếp với HPS thông qua block FPGA to HPS.

Như vậy ta có thể thiết kế một hệ thống như sau:



Hình 44: HPS đóng vai trò thực hiện chức năng đọc dữ liệu từ camera



Hình 45: Kết nối giữa HPS và FPGA

Đối với thực hiện mô hình CNN dùng thuần FPGA sẽ rất khó để thực hiện vì vậy ta cần HPS là một hard processer system đóng vai trò quản lý và điều khiển việc thực hiện mô hình, Phần system design của phần FPGA domain ta sẽ thiết kế một bộ thực hiện tích chập với tốc độ cao, Phần HPS sẽ thực hiện các công việc như đọc dữ liệu từ camera, lưu vào bộ nhớ, chuyển dữ liệu đến đầu vào của bộ tính tích chập ở FPGA để thực hiện việc tính toán và HPS chỉ cần load dữ liệu từ đầu ra của bộ tích chập ở FPGA và lưu lại vào bộ nhớ đệm. Vì HPS không tự mình thực hiện việc tính tích chập mà việc tính tích chập được thực hiện bởi FPGA nên hệ thống có khã năng đáp ứng được tốc độ cao.

TÀI LIỆU THAM KHẢO

1. Dong Wang, Ke Xu and Diankun Jiang, (2017), *PipeCNN: An OpenCL-Based Open-Source FPGA Accelerator for Convolution Neural Networks*
2. Intel, (truy cập ngày 28/08/2023), *Map HPS IP Peripheral Signals to FPGA Interface,* truy cập tại https://www.intel.com/content/www/us/en/support/programmable/support-resources/design-examples/horizontal/exm-hps-fpga-interface.html
3. Tianxu Yue (2021), *Convolutional Neural Network FPGA-accelerator on Intel DE10- Standard FPGA*