● نکات پیاده سازی و کدهای زده شده:

activationFunction :

این ماژول یک عدد که به شکل ۱۶ بیتی با فرمت اندازه علامت است و ۶ بیت اعشاری و ۱۰ بیت صحیح دارد را به عنوان ورودی می‌گیرد، علاوه بر این ورودی یک سیگنال به نام ready را نیز می‌گیرد که نشان می‌دهد که آماده‌ی انجام عملیات هستم یا نه. این تابع فعال سازی به این شکل عمل می‌کند که در صورتی که ورودی کوچکتر از ۰ باشد ۰ را به عنوان خروجی می‌دهد و اگر بزرگتر و یا مساوی صفر باشد خود عدد را به عنوان خروجی می‌دهد. نحوه‌ی انجام مقایسه‌ی استفاده از مقایسه‌کننده‌ای از که از کنار هم قرار گرفتن ۱۶ ماژول مقایسه‌کننده‌ی ۱ بیتی ایجاد شده است. درستی این ماژول را با استفاده از تست بنچ activeTB بررسی کردیم

adder :

این ماژول ۲ عدد با تعداد بیت‌های برابر را که با فرمت اندازه علامت نمایش داده می‌شوند به عنوان ورودی می‌گیرد و جمع این ۲ را به عنوان خروجی می‌دهد.

برای جمع کردن اعدادی که با فرمت اندازه علامت نمایش داده می‌شوند به این شکل عمل می‌کنیم که علامت و اندازه‌های اعداد را به طور جداگانه بررسی می‌کنیم، چندین حالت مختلف می‌تواند وجود داشته باشد:

- ۲ عدد منفی باشند:

اندازه‌های ۲ عدد را با هم جمع می‌کنیم سپس بیت علامت را برابر با ۱ گذاشته تا منفی بودن عدد را نشان دهد

- ۲ عدد مثبت باشند:

اندازه‌های ۲ عدد را با هم جمع می‌کنیم سپس بیت علامت را برابر با ۰ گذاشته تا مثبت بودن عدد را نشان دهد

- یک عدد مثبت و دیگری منفی:

* عدد مثبت بزرگتر باشد

اندازه‌ی عدد مثبت را از اندازه‌ی عدد منفی کم کرده و بیت علامت را ۰ می‌گذاریم.

* عدد منفی بزرگتر باشد

اندازه‌ی عدد منفی را از اندازه‌ی عدد مثبت کم کرده و بیت علامت را ۱ می‌گذاریم.

برای پیاده‌سازی منطق توضیح داده شده از یک مقایسه‌کننده استفاده کردیم و کد را به صورت زیر نوشتم:

```
process (a, b, a_gt_b)
    variable add_res : unsigned(width - 2 downto 0);
begin
    if(a(width - 1) = '1' and b(width - 1) = '1') then
        add_res := unsigned(a(width - 2 downto 0)) + unsigned(b(width - 2 downto 0));
        res <= std_logic_vector('1' & add_res(width - 2 downto 0));
    elsif (a(width - 1) = '0' and b(width - 1) = '1') then
        if(a_gt_b='1')then
            add_res := unsigned(a(width - 2 downto 0)) - unsigned(b(width - 2 downto 0));
            res <= std_logic_vector('0' & add_res(width - 2 downto 0));
        elsif(a_eq_b='1')then
            res <= (others=>'0');
        else
            add_res := unsigned(b(width - 2 downto 0)) - unsigned(a(width - 2 downto 0));
            res <= std_logic_vector('1' & add_res(width - 2 downto 0));
        end if;
    elsif (a(width - 1) = '1' and b(width - 1) = '0') then
        if(a_gt_b='1')then
            add_res := unsigned(a(width - 2 downto 0)) - unsigned(b(width - 2 downto 0));
            res <= std_logic_vector('1' & add_res(width - 2 downto 0));
        elsif(a_eq_b='1')then
            res <= (others=>'0');
        else
            add_res := unsigned(b(width - 2 downto 0)) - unsigned(a(width - 2 downto 0));
            res <= std_logic_vector('0' & add_res(width - 2 downto 0));
        end if;
    else
        add_res := unsigned(a(width - 2 downto 0)) + unsigned(b(width - 2 downto 0));
        res <= '0' & std_logic_vector(add_res(width - 2 downto 0));
    end if;

end process;
```

: Controller

کنترلر این پروژه مشابه فازهای قبل مربوط به هر نورون است یعنی ما به این شکل عمل کردیم که هر لایه خروجی‌ای را مبنی بر اتمام کار لایه می‌دهد که این خروجی را به عنوان ورودی لایه‌ی بعد در نظر می‌گیریم که اجازه‌ی شروع کار را به آن لایه می‌دهد، با توجه به این نکته و این مسئله که لایه‌های ما از تعدادی نورون تشکیل شده اند می‌توانیم کنترلر را فقط برای نورون بنویسیم و با این وجود تمام شبکه عصبی منظم و درست کار خواهند کرد.

برای تعریف state های مختلف از نوع داده‌ی enum استفاده کردیم تا بتوانیم به جای اعداد اسم‌هایی را برای این حالت‌ها قرار دهیم و فهمیدن کد ساده‌تر شود.

جابجایی بین state ها با لبه‌ی بالا رونده‌ی کلاک اتفاق می‌افتد.

یکی از خروجی‌های این ماژول آدرس نام دارد که زمانی که شبکه عصبی ما چندین ورودی دارد نشان می‌دهد که کدام ورودی را باید بخوانیم و محاسبات را برای آن انجام دهیم.

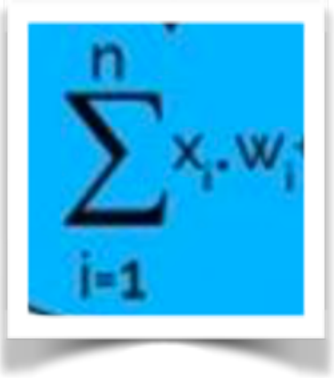
شکل کلی این کنترلر در ادامه آمده است.

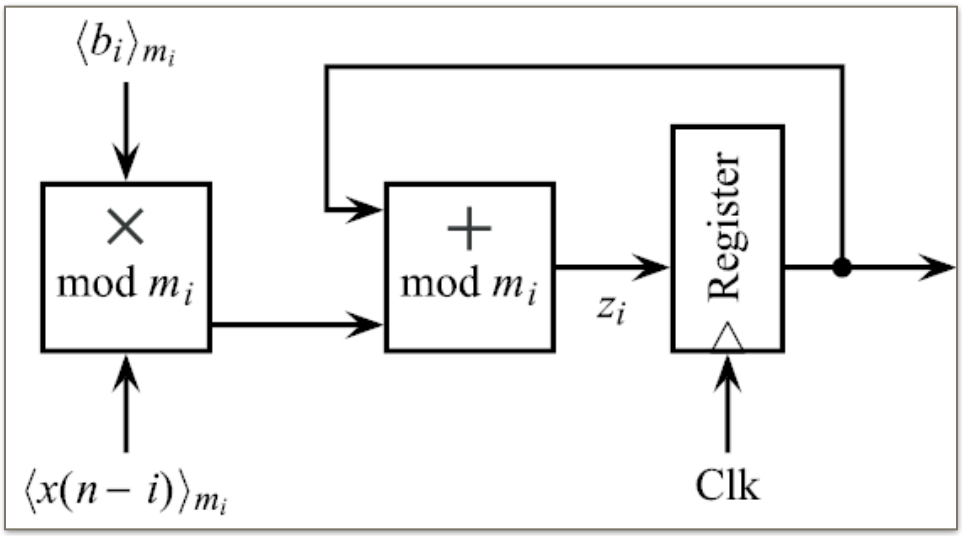
: Input_selector

این ماژول وظیفه دارد که لیست کل ورودی‌های سیستم را بگیرد و با توجه به آدرس که یکی دیگر از ورودی‌های این ماژول است ورودی‌ای را که باید به شبکه داده شود انتخاب می‌کند و به عنوان خروجی می‌دهد.

:mac

نام کامل این واحد: multiply and accumulate این واحد باید چیزی مشابه عبارت زیر را محاسبه کند، ساختار کلی آن هم نشان داده شده است.





کد ما نیز دقیقا به شکل زیر دقیقا همین ساختار را دنبال می‌کند:

```
m1: mult
    generic map (width => width, point => point)
    port map(data1_in, data2_in, multiplication_value);
a1: adder
    generic map (width => width, point => point)
    port map(multiplication_value, accumulated_value, added_value);
r1: reg
    generic map (width => width, point => point)
    port map(added_value, clk, load, rst, accumulated_value);
data_out <= accumulated_value;
```

جمع‌کننده را بالاتر توضیح دادیم ولی توضیحات تکمیلی زیر را نیز درباره‌ی جمع‌کننده و ضرب کننده می‌توان ارائه داد:

در بخش جمع‌کننده که یکی از componentهای بخش MAC است، با توجّه به بیت علامت هر یک از دو عددی که با هم جمع می‌شوند، عملیات مورد نیاز برای جمع انجام شده است. در فایل adder در صورتی که علامت هر دو عدد با هم برابر باشد سایر بیت‌ها را با هم جمع می‌کنیم و با علامت مشترک در res می‌ریزیم (خطوط ۴۵ تا ۴۷ و ۶۸ تا ۷۰). برای این‌که بتوانیم تشخیص دهیم در صورتی که یکی از اعداد منفی و دیگری مثبت بود کدام‌یک را از دیگری کم کنیم از یک nibble comparator استفاده کرده‌ایم که برای طرّاحی آن، طبق مباحثی که در کلاس مطرح شده بود، از چند component از نوع bit comparator استفاده کردیم که کدهای آن به‌ترتیب در فایل های n_bit_comparatoe و bit_comparator موجود است. حال در صورتی که علامت دو عدد برابر نبود سه حالت را در نظر می‌گیریم، اگر خروجی مقایسه‌کننده‌ای که ورودی‌اش دو عدد ما است (خطوط ۳۸ تا ۴۰) a_eq_b بود، res را برابر با ۱۶ بیت صفر می‌کنیم (خطوط ۵۲ و ۵۳ و همچنین ۶۲ و ۶۳)، اگر خروجی a_gt_b بود a - b را (که طبعاً به‌صورت unsigned جمع می‌شود) در res می‌ریزیم (خطوط ۴۹ تا ۵۱ و همچنین ۵۹ تا ۶۱ و برای حالتی که خروجی a_lt_b بود نیز b - a را در res می‌ریزیم. در همه‌ی حالات در صورت overflow مقدار اضافی را بیرون می‌ریزیم و فقط n (=۱۶) بیت سمت راست باقی می‌ماند. (قطعاً علامت صحیح خواهد بود.)

در بخش ضرب‌کننده که دیگر component بخش MAC است، به این صورت عمل می‌کنیم که برای مشخّص کردن بیت علامت نتیجه، بیت‌های علامت دو عدد ورودی را با هم xor می‌کنیم، که به این معناست که اگر هر دو علامت با هم یکی بود (منفی یا مثبت) نتیجه مثبت می‌شود و در غیر این صورت منفی. برای بخش عددی نتیجه نیز حاصل‌ضربِ unsigned دو عدد را (به‌جز بیت علامت) حساب می‌کنیم و این دو بخش را در نهایت با هم concatenate می‌کنیم تا به نتیجه‌ی نهایی برسیم. (خطوط ۱۴ تا ۲۲ از فایل mult) در واقع خروجی نهایی این تابع از point + 2 -width تا point از همه‌ی بیت‌های حاصل‌ضرب خواهد بود. (خطّ ۲۱ از همان فایل)

**** نکات پیاده سازی:**

۱. یک تفاوت عمده این پروژه با پروژه قبل در قسمت نورون اضافه شدن bias که در نهایت باید با حاصل mac جمع شود، برای پیاده سازی این موضوع از یک واحد جمع کننده‌ی اضافه استفاده کردیم

۲. موضوع دیگر تعداد زیاد نورون‌ها در این پروژه و نیاز به کار هماهنگ آن‌ها است، به این معنی که تا زمانی که تمام نورون‌های یک لایه کار خود را شروع نکرده اند لایه‌ی دیگری نباید کار خود را شروع کند چون در آن زمان داده‌ای که در ورودی‌ها قرار دارد قابل اعتماد نیست و درست نیست. به همین دلیل برای هر لایه سیگنال‌های شروع و پایان قرار دادیم که سیگنال پایان هر قسمت شروع قسمت بعد بود و در صورتی پایان یک می‌شد که تمام نورون های آن لایه کار خود را تمام کرده باشند

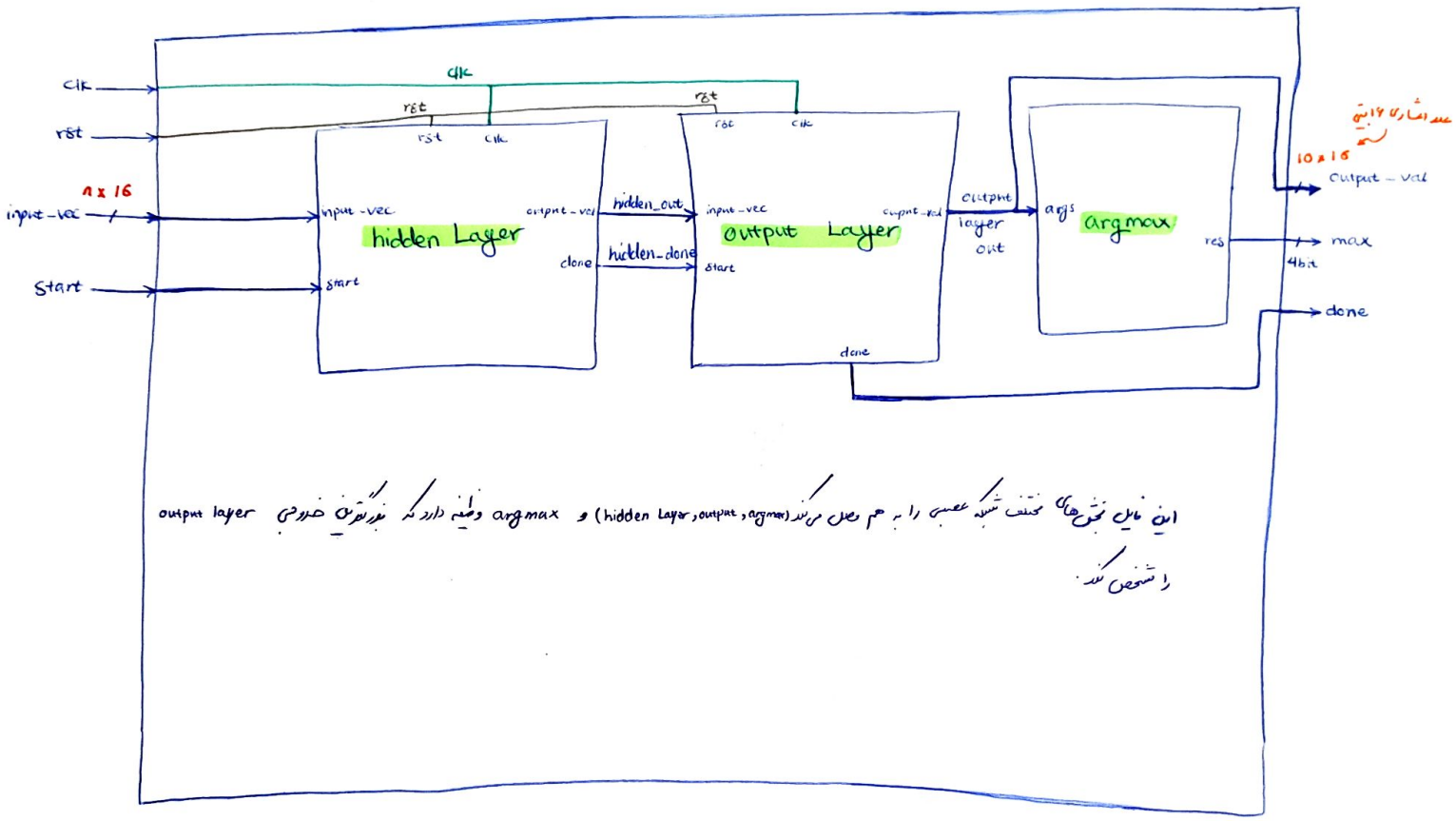
۳. در این پروژه وزن‌ها و bias ها را به عنوان ورودی تابع‌ها نمی‌دادیم و مقادیر از فایل های حاوی این اطلاعات خوانده می‌شد. این کار را با استفاده از توابع تعریف شده در فایل nmn_types انجام دادیم.

rom_init: این تابع اسم فایلی که اطلاعات در آن است را به همراه طول فایل می‌گیرد و یک آرایه از مقادیر خوانده شده از فایل را به عنوان خروجی می‌دهد

rom62_init و rom20_init نیز کارهای مشابهی انجام می‌دهد با این تفاوت که اندازه‌ی داده‌هایی که می‌خوانند متفاوت است یعنی مقلا یکی یک آرایه‌ی n *20 و دیگری یک آرایه‌ی n*62 را مقدار دهی اولیه می‌کند.

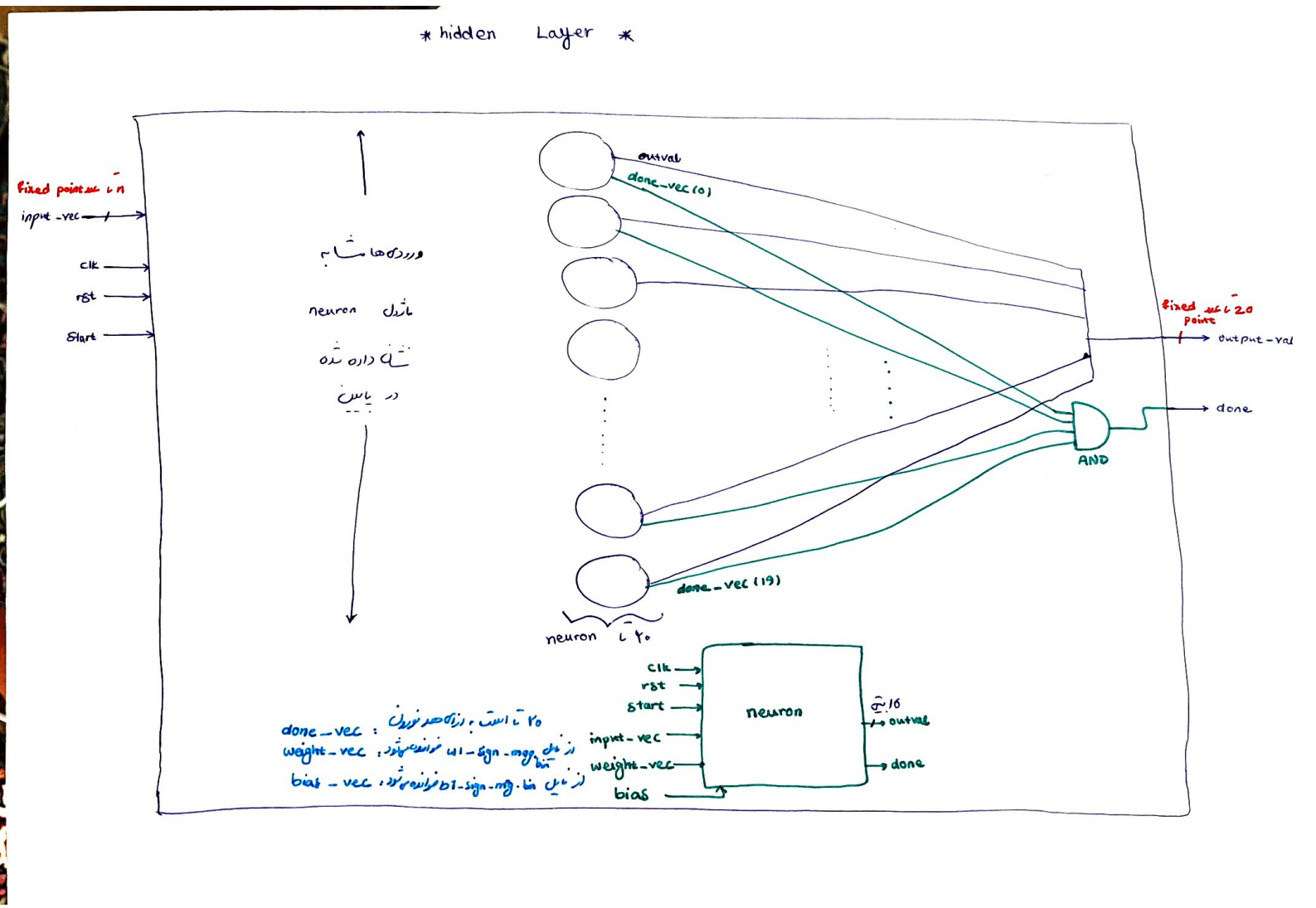
بلاک دیاگرام طراحی

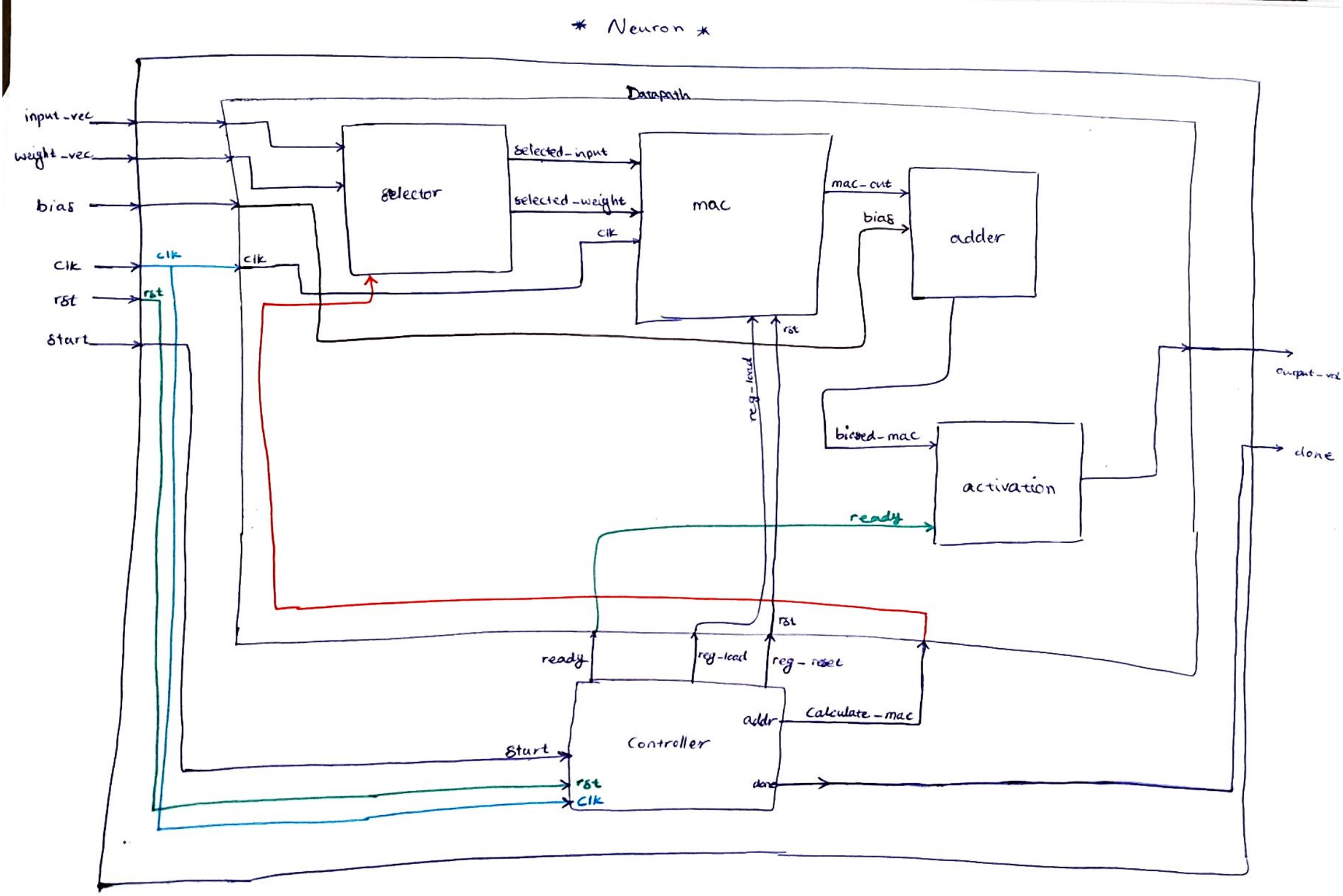
* MLP *

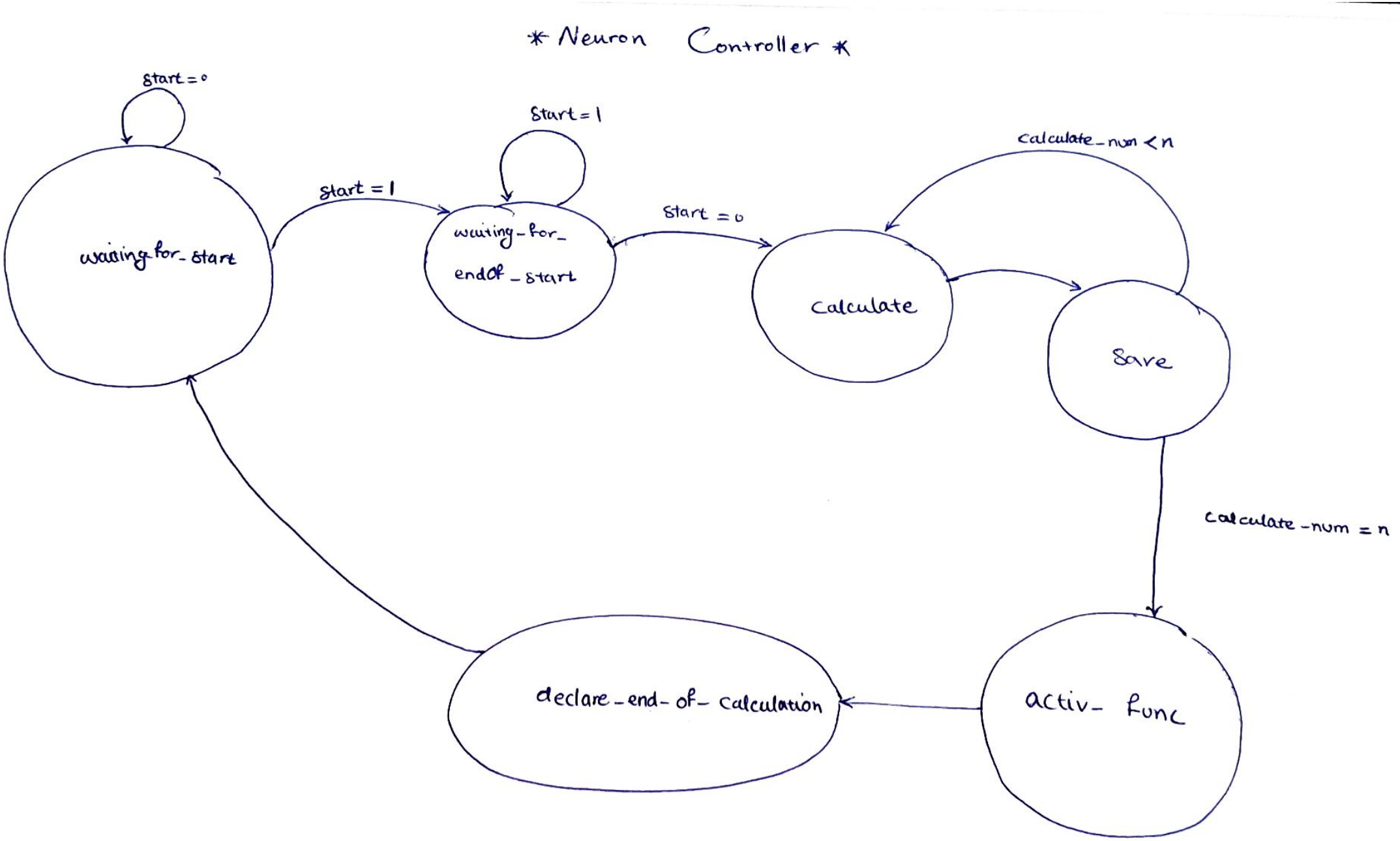


عدد شاره ۱۶ بیتی

این فایل نقش‌های مختلف شبکه عصبی را به هم وصل می‌کند (hidden Layer, output, argmax) و argmax وظیفه دارد که بزرگترین خروجی output layer را تشخیص دهد.


* hidden Layer *
fixed point ۱۶ بیتی
input_vec
clk
rst
start
ورودی‌های شبکه
neuron لایه‌ی
شبکه داده شده
در پایین
out_val
done_vec(0)
done_vec(19)
neuron ۲۰ تا
AND
fixed point ۱۶ بیتی ۲۰ تا
output_val
done
clk
rst
start
input_vec
weight_vec
bias
neuron
out_val
done
done_vec : ۲۰ تا است و زمانی که هر نورون
weight_vec : از فایل mif-sign-mag خوانده می‌شود
bias_vec : از فایل mif-sign-mag خوانده می‌شود


* Neuron *
Datapath
input_vec
weight_vec
bias
clk
rst
start
selector
selected_input
selected_weight
clk
mac
mac_out
bias
adder
reg_load
rst
biased_mac
activation
output_vec
done
ready
rst
ready
reg_load
reg_reset
addr
Calculate_mac
Controller
start
rst
clk
done


* Neuron Controller *
start = 0
waiting_for_start
start = 1
Start = 1
waiting_for_
endOf_start
start = 0
calculate_num < n
calculate
Save
calculate_num = n
activ_ func
declare_end_of_ calculation

* output_Layer *