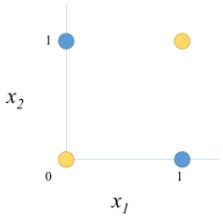
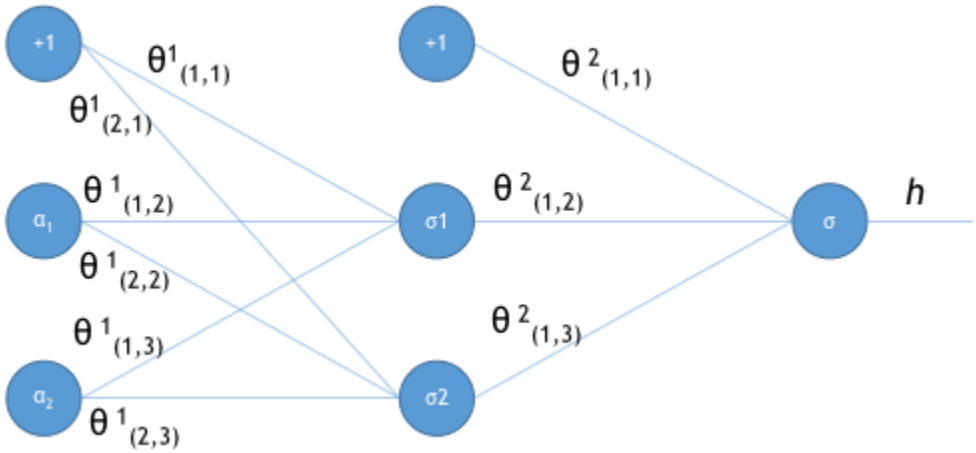
VLSI Project Report

Ho Yu Hin (3035276696)

1. Abstract:

In this project, I will implement an artificial neural network by a FPGA using Verilog HDL to solve an XOR problem, which is a classic problem in the Artificial neural network (ANN) research. It is a classification problem for which the expected output is known, the output is 0 if two inputs are the same; the output is 1 if two inputs are different, as shown in the figure at the right. Supervised learning will be used to train the neural network since we know the actual output. This is the topology of the network.





Neural networks are composed of nodes connected by directed link. Each link has a weight associated with it that determines the strength and the sign of connection. The network is arranged in layers: input layer, hidden layers and output layer.
In every hidden layer and output layer, the output is passed into an activation function to produce a value between 0 and 1, which is the probability of giving the desired output.
In this project, I will use a sigmoid function defined as below.

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

The calculated output would be different from the expected output. Hence, backpropagation is needed to tune the weight value and the bias. We use a commonly used loss function to measure the size of the error.

$$L(a, y) = -(y\log(a) + (1 - y)\log(1 - a))$$

2. High level programming language implementation (Both prediction and backpropagation)

I implement the XOR neural network in python.

First, import numpy library for matrix calculation.

After that, define two methods - sigmoid and xor_nn. For simplicity, we ignore the bias and thus the equation is $y = w_1 x_1 + w_2 x_2$. In the implementation, w_1 and w_2 are called THETA1 and THETA2 respectively. THETA1 and THETA2 are basically a 2*3 matrix and 1*3 matrix respectively. For THETA1, there are two rows because the output points to two variables in the hidden layer. For THETA2, there is one row only because the output points to one variable which is the output. They both consist of three columns, 1 which is the boundary and two input variables.

There are 6 parameters in the function input. XOR consists of the input and output. THETA1 and THETA2 are the weight. Init_w means whether we want to initialize the THETA arrays. Learn means whether we are training the neural network. Alpha is the learning rate.

The code is as follows:

```python
import numpy as np
import math

def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))

def xor_nn(XOR, THETA1, THETA2, init_w=0, learn=0, alpha=0.01):
    if init_w == 1:
        THETA1 = 2*np.random.random([2,3]) - 1  # 2*3 matrix. 2 since it point to hiddenVar 1 and 2. 3 since 1, input1, input2
        THETA2 = 2*np.random.random([1,3]) - 1  # 1*3 matrix. 1 since it point to output. 3 since 1, hiddenVar1 and 2

    T1_DELTA = np.zeros(THETA1.shape)
    T2_DELTA = np.zeros(THETA2.shape)
        # count of x
    m = 0
        # loss
    J = 0.0

    for x in XOR:
        A1 = np.vstack(([1], np.transpose(x[0:2][np.newaxis])))
        Z2 = np.dot(THETA1, A1)
        A2 = np.vstack(([1], sigmoid(Z2)))
        Z3 = np.dot(THETA2, A2)
        h = sigmoid(Z3)
  # x[2] = output, sum the loss function for different input
        J = J + (x[2] * math.log(h[0])) + ((1 - x[2]) * math.log(1 - h[0]));
        m = m + 1;

        if learn == 1:
                # dL/dz = a-y
            delta3 = h - x[2]
            print("delta3")
            print(delta3)
            delta2 = (np.dot(np.transpose(THETA2), delta3) * (A2 * (1 - A2)))[1:]
            print("delta2")
            print(delta2)
                # dL/dw1
            T2_DELTA = T2_DELTA + np.dot(delta3, np.transpose(A2))
                # dL/dw2
            T1_DELTA = T1_DELTA + np.dot(delta2, np.transpose(A1))
            print("T1_del")
            print(T1_DELTA)
        else:
            #print('Hypothesis: ');
            print(h)
       # finally get the total loss
    J = J / -m
```

```
48
49          # b' = b-alpha(a-y)
50      if learn == 1:
51          THETA1 = THETA1 - (alpha * (T1_DELTA / m))
52          THETA2 = THETA2 - (alpha * (T2_DELTA / m))
53      else:
54          print(J)
55
56      return (THETA1, THETA2)
57
58  XOR = np.array([[0,0,0], [0,1,1], [1,0,1], [1,1,0]])
59  THETA1, THETA2 = xor_nn(XOR, 0, 0, 1, 1, 0.01)
60  THETA1, THETA2 = xor_nn(XOR, THETA1, THETA2, 0, 1, 0.01)
```

Result:

10,000 iterations

```
In [13]: runfile('C:/Users/Ho Yu Hin/Desktop/vlsitest.py', wdir='C:/Users/Ho Yu
Hin/Desktop')
Reloaded modules: vlsi
('Elapsed time ', 1.8230053142144698)
Neural Network Program
Inputs:  0, 0 Output: [[0.4681824]] Expected Output: 0 Error: [[0.4681824]]
Inputs:  0, 1 Output: [[0.50751037]] Expected Output: 1 Error: [[0.49248963]]
Inputs:  1, 0 Output: [[0.49478708]] Expected Output: 1 Error: [[0.50521292]]
Inputs:  1, 1 Output: [[0.52794612]] Expected Output: 0 Error: [[0.52794612]]
```

100,000 iterations

```
In [12]: runfile('C:/Users/Ho Yu Hin/Desktop/vlsitest.py', wdir='C:/Users/Ho Yu
Hin/Desktop')
Reloaded modules: vlsi
('Elapsed time ', 18.678037632944182)
Neural Network Program
Inputs:  0, 0 Output: [[0.0161629]] Expected Output: 0 Error: [[0.0161629]]
Inputs:  0, 1 Output: [[0.97718102]] Expected Output: 1 Error: [[0.02281898]]
Inputs:  1, 0 Output: [[0.98602965]] Expected Output: 1 Error: [[0.01397035]]
Inputs:  1, 1 Output: [[0.01379675]] Expected Output: 0 Error: [[0.01379675]]
```

1,000,000 iterations

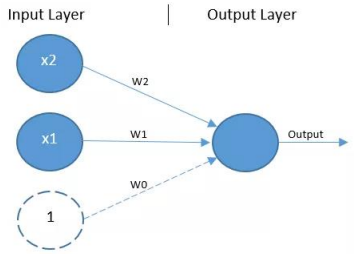
```
In [16]: runfile('C:/Users/Ho Yu Hin/Desktop/vlsitest.py', wdir='C:/Users/Ho Yu
Hin/Desktop')
Reloaded modules: vlsi
('Elapsed time ', 183.85700570739215)
Neural Network Program
Inputs:  0, 0 Output: [[0.00119796]] Expected Output: 0 Error: [[0.00119796]]
Inputs:  0, 1 Output: [[0.9990804]] Expected Output: 1 Error: [[0.0009196]]
Inputs:  1, 0 Output: [[0.99908285]] Expected Output: 1 Error: [[0.00091715]]
Inputs:  1, 1 Output: [[0.00106543]] Expected Output: 0 Error: [[0.00106543]]
```

As we can see, the output is completely unacceptable for 10000 iterations. The output is satisfactory for 100000 iterations or above.

Reference:

https://aimatters.wordpress.com/2016/01/11/solving-xor-with-a-neural-network-in-python/

3. Verilog implementation



For simplicity, no hidden layer is used in this model.

The steps are as follows.

1. Calculate $\mathrm{z} = \mathrm{w}_1 x_1 + w_2 x_2 + b$
2. Pass z to the sigmoid function to get $\mathrm{a} = \sigma(\mathrm{z})$
3. Calculate the loss by $\mathrm{L(a,y)} = -(\mathrm{y}\log(\mathrm{a}) + (1-\mathrm{y})\log(1-\mathrm{a}))$
4. Backward propagation to get the new $\mathrm{w}_1', \mathrm{w}_2'$ and b' with the following equations

$$\mathrm{w}_1' = w_1 - \alpha x_1 (a - y)$$
$$\mathrm{w}_2' = w_2 - \alpha x_2 (a - y)$$
$$b' = b - \alpha (a - y)$$

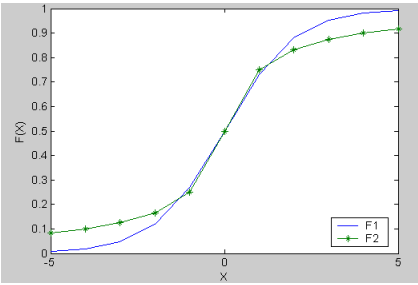
   Where α is the learning rate, we can set $\alpha = 0.01$.
5. Repeat 1-4 for a certain number of times or until the loss is smaller than a certain value.

Function / Building block required:

1. Arithmetic of fixed point number
2. Sigmoid calculation

   Since it is uneasy to calculate the exponential function in the sigmoid function, an approximation can be used. As Thamer M.Jamel and Ban M.Khammas suggested in the 13th Scientific Conference of AI-Ma'moon University College,

   $\mathrm{F2(x)} = \frac{1}{2}\left[\frac{x}{1+|x|} + 1\right]$ is a good approximation of the sigmoid function for small x.



3. Loss function calculation

3.1 Arithmetic of fixed point number

Format of fixed point number

|1|<- N-Q-1 bits ->|<--- Q bits -->|

The first bit is the sign bit, where 0 means positive and 1 means negative.

Here I use 16 bits to represent the integer part and 15 bits to represent the decimal part.

For example: 0 0000 0000 0000 0001 110 0000 0000 0000 is 1+0.5+0.25=1.75.

In this part, I make use of the following library for the arithmetic of fixed point number.

https://opencores.org/project/verilog_fixed_point_math_library/overview

I modified the codes, for example adding clock input and done output to enhance the functionality.

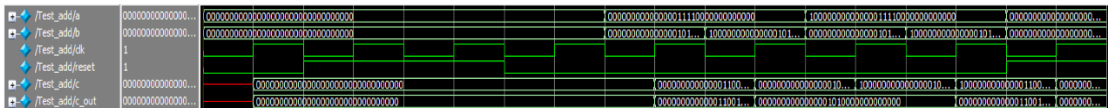
Addition and Subtraction

Part of the codes in test bench:

```
initial begin
    // Initialize Inputs
    a[31:0] = 0;
    b[31:0] = 0;
    clk = 0;
    reset = 0;

    // Wait 100 ns for global reset to finish
    #(`CLK) reset = 1;
    #(`CLK);
    #(`CLK) reset = 0;

    // Test of 3.75 + 2.5 = 6.75
    #(`CLK) a[30:15] = 3; b[30:15] = 2;a[31] = 0; a[14] = 1; a[13] = 1; b[31] = 0;b[14] = 1;b[13] = 0;

    // Test of 3.75 - 2.5 = 1.25
    #(`CLK) a[30:15] = 3; b[30:15] = 2;a[31] = 0; a[14] = 1; a[13] = 1; b[31] = 1;b[14] = 1;b[13] = 0;

    // Test of -3.75 + 2.5 = -1.25
    #(`CLK) a[30:15] = 3; b[30:15] = 2;a[31] = 1; a[14] = 1; a[13] = 1; b[31] = 0;b[14] = 1;b[13] = 0;

    // Test of -3.75 - 2.5 = -6.75
    #(`CLK) a[30:15] = 3; b[30:15] = 2;a[31] = 1; a[14] = 1; a[13] = 1; b[31] = 1;b[14] = 1;b[13] = 0;

    #(`CLK) reset = 1;a[31:0] = 0; b[31:0] = 0;

    #(`CLK) $finish;
    end
```

Result:

```
# time=                   50: a=0.00 b=0.00 s=0.00
# time=                  150: a=0.00 b=0.00 s=0.00
# time=                  250: a=0.00 b=0.00 s=0.00
# time=                  350: a=0.00 b=0.00 s=0.00
# time=                  450: a=3.75 b=2.50 s=6.25
# time=                  550: a=3.75 b=-2.50 s=1.25
# time=                  650: a=-3.75 b=2.50 s=-1.25
# time=                  750: a=-3.75 b=-2.50 s=-6.25
# time=                  850: a=0.00 b=0.00 s=0.00
# ** Note: $finish    : D:/intelFPGA_lite/VLSI/Test_add.v(68)
#    Time: 900 ns  Iteration: 0  Instance: /Test_add
```



Explanation:

There are 4 test cases:

1. Positive, positive
2. Positive, negative
3. Negative, positive
4. Negative, negative

All test cases give the expected result.

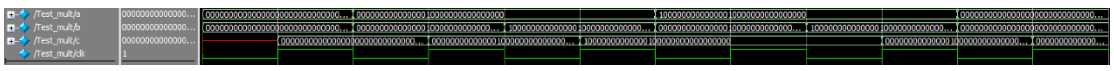
Multiplication

Part of the codes in test bench:

```
initial begin
    // Initialize Inputs
    a[31:0] = 0;
    b[31:0] = 0;
    clk = 0;

    // Test of 2 * 2 = 4
    #(`CLK) a[30:15] = 2; b[30:15] = 2;a[31] = 0; b[31] = 0;

    // Test of 2 * - 2 = -4
    #(`CLK) a[30:15] = 2; b[30:15] = 2;a[31] = 0; b[31] = 1;

    // Test of -2 * 2 = -4
    #(`CLK) a[30:15] = 2; b[30:15] = 2;a[31] = 1; b[31] = 0;

    // Test of -2 * -2 = 4
    #(`CLK) a[30:15] = 2; b[30:15] = 2;a[31] = 1; b[31] = 1;

    #(`CLK) a[31:0] = 0; b[31:0] = 0;
    #(`CLK) $finish;
    end
```

Result:

```
# time=                   50: a=0.00 b=0.00 s=0.00
# time=                  150: a=2.00 b=2.00 s=4.00
# time=                  250: a=2.00 b=-2.00 s=-4.00
# time=                  350: a=-2.00 b=2.00 s=-4.00
# time=                  450: a=-2.00 b=-2.00 s=4.00
# time=                  550: a=0.00 b=0.00 s=0.00
# ** Note: $finish    : D:/intelFPGA_lite/VLSI/Test_mult.v(43)
#    Time: 600 ns  Iteration: 0  Instance: /Test_mult
```



Explanation:

There are 4 test cases:

1. Positive, positive
2. Positive, negative
3. Negative, positive
4. Negative, negative

All test cases give the expected result.

Division

Part of the codes in test bench:

```
initial begin
    // Initialize Inputs
    top = 2.5;
    bottom = -2.25;

    conv_rational(top,dividend);
    conv_rational(bottom,divisor);

    start = 1;
    clk = 0;

    // Wait 100 ns for global reset to finish
    #100;

    // Add stimulus here
    start = 0;

    #366;
    conv_fixed(quotient,result);
    $display("%f / %f = %f",top,bottom,result);
end
```

Result:

```
# 2.500000 / 2.250000 = 1.111115
# 2.500000 / -2.250000 = -1.111115
# -2.500000 / 2.250000 = -1.111115
# -2.500000 / -2.250000 = 1.111115
```

Explanation:

There are 4 test cases:

1. Positive, positive
2. Positive, negative
3. Negative, positive
4. Negative, negative

The expected output should be +1.111111 or -1.111111, there is a small difference due to precision error.

After testing the fixed-point arithmetic library, it can be used to build different function.

3.2 Weighting function implementation

As mentioned above, I will implement function for calculating $\mathrm{z} = \mathrm{w}_1 x_1 + w_2 x_2 + b$

Procedure:

1. Compute $\mathrm{w}_1 x_1$ and $w_2 x_2$ parallelly.
2. Compute $\mathrm{out}_1 = \mathrm{w}_1 x_1 + w_2 x_2$
3. Compute $\mathrm{z} = \mathrm{out}_1 + b$

Code:

```
initial done = 0;
    qmult #(15,32) uutMult1 (
    .a(w1),
    .b(x1),
    .c(mult1_w),
    .clk(multclk)
);

    qmult #(15,32) uutMult2 (
    .a(w2),
    .b(x2),
    .c(mult2_w),
    .clk(multclk)
);

    qadd #(15,32) uutAdd1 (
    .a(mult1_w),
    .b(mult2_w),
    .c(out1),
    .clk(add1clk),
    .reset(add1rst)
);
    qadd #(15,32) uutAdd2 (
    .a(out1),
    .b(b),
    .c(out2),
    .clk(add2clk),
    .reset(add2rst)
);
```

```
prepared <= 1'b1;
x1 <= x1_i;
w1 <= w1_i;
x2 <= x2_i;
w2 <= w2_i;
b <= b_i;
#50
multclk = 1;
#100;
add1clk = 1;
#50;
add2clk = 1;
#50;
done = 1;
```

Explanation:

The code on the left shows the connection of modules.

The code on the right shows main logic. We first store the input in registers. Delay a bit. Perform the multiplication. Delay a bit. Perform the addition. Delay a bit. Perform the addition.

Testing & Results:

Here we set the $w_1 = 0.25, w_2 = 0.75$ and $b = 0.5$.

Test for input 00,01,10 and 11.

```
#  0.250000 0.000000 + 0.750000 0.000000 + 0.500000 => 0.500000
#  0.250000 1.000000 + 0.750000 0.000000 + 0.500000 => 0.750000
#  0.250000 0.000000 + 0.750000 1.000000 + 0.500000 => 1.250000
#  0.250000 1.000000 + 0.750000 1.000000 + 0.500000 => 1.500000
```

It gives the expected output.

3.3 Sigmoid function implementation:

As mentioned above, I will implement the approximation of sigmoid

$$\frac{1}{2}\left[\frac{x}{1+|x|}+1\right]$$

In Verilog.

Procedure:

1. Calculate the |x| by setting the sign bit of x to 0
2. Calculate 1+|x| by adding 1 to |x| using the qadd module
3. Calculate $\frac{x}{1+|x|}$ using the qdiv module
4. Calculate $\left[\frac{x}{1+|x|}+1\right]$ using the qadd module
5. Calculate $\frac{1}{2}\left[\frac{x}{1+|x|}+1\right]$ using the qmul module

Code:

```
82  always @( posedge clk ) begin
83  // Stage 1: started and ready for preparing the dividend and divisor
84  if( !prepared && start ) begin
85      prepared <= 1'b1;
86      dividend <= x;
87      divisor[30:0] <= x[30:0];
88      //abs
89      divisor[31]<=0;
90
91
92      addOne[31:0]=0;
93      addOne[15]=1;
94      addclk = 0;
95      addrst = 0;
96      #50;
97      addclk = 1;
98      #50;
99  end
100 // Stage 2: prepared and ready for perform division
101 else if (prepared && !divDone) begin
102
103         #100
104         divStart = 1;
105         #100;
106
107         divStart = 0;
108         #366;
109         divDone <= divDone_w;
110 end
111 // Stage 3: adding the 1 and multiply with 0.5, reset parameters
112 else if (divDone && prepared) begin
113     #50;
114     out=outtest;
115     mulHalf[31:0]=0;
116     mulHalf[14]=1;
117     #50;
118     multClk = 1;
119     #100;
120     done = 1;
121     #50;
122     done = 0;
123     prepared <=0;
124     divDone <=0;
125 end
```

Explanation:

The code is straightforward. Please refer to the comment in the code.

Testing & Results:

Four test cases are used here, which are 0.25, -0.25, 2.25 and -2.25.

The result is tabulated as follows.

Input	sigmoid	approximation	absolute error
0.25	0.562177	0.600006	0.037829
-0.25	0.437823	0.399994	0.037829
2.25	0.904650	0.846130	0.058520
-2.25	0.095349	0.153839	0.058490

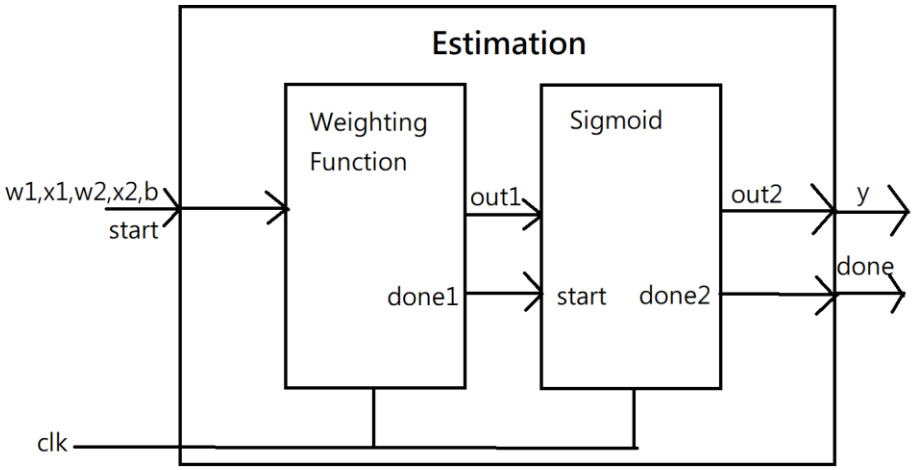
```
#  0.250000 => 0.600006
#  -0.250000 => 0.399994
#  2.250000 => 0.846130
```

```
#  -2.250000 => 0.153839
```

We can see the function can approximate sigmoid well. The error is smaller when the input is closer to 0.

After creating the weighting function and sigmoid function, we can now combine these two functions to form a function called forward.

3.4 Estimation implementation



The above figure describes the structure of the function.

```
calz uut1(
.w1_i(w1),
.x1_i(x1),
.w2_i(w2),
.x2_i(x2),
.b_i(b),
.y(y_out1),
.clk(clk),
.start(startSignal),
.complete(done1)
);


sigmoid uut2(
.x(y_out1),
.y(y_out2),
.clk(clk),
.start(done1),
.complete(done2)
);
```

This describes the connection between these two modules.

Test case and result:

```
#  sigmoid(0.250000 1.000000 + 0.750000 1.000000 + 0.500000) => 0.799988
# ** Note: $finish    : D:/intelFPGA_lite/VLSI/TestForward.v(61)
#    Time: 8 us  Iteration: 0  Instance: /TestForward
```

The expected output should be 0.8 if we use the approximation of the sigmoid

function. There is a small difference due to precision error.

3.5 Backward Propagation implementation

In this part, I will implement the following formula to calculate the new weight

$$w_i' = w_i - \alpha x_i(a - y)$$

Where α is the learning rate.

Procedure:

1. Setting the sign bit of α and y to 1 for the subtraction
2. Compute $-\alpha x_i$ and $a - y$ parallelly
3. Compute $-\alpha x_i(a - y)$
4. Compute $w_i' = w_i - \alpha x_i(a - y)$

Code:

```
prepared <= 1'b1;
w <= w_i;
x <= x_i;
a <= a_i;
y[31] <= 1;
y[30:0] <= y_i[30:0];
alpha[31] <= 1;
alpha[30:0] <= alpha_i[30:0];

#50
mult1clk = 1;
add1clk = 1;
#100;
mult2clk = 1;
#50;
add2clk = 1;
#50;
done = 1;
```

Explanation:

This code is similar to the code weighting function.

Testing & Results:

Due to the similarity of the code, I only present one test case.

Here we set the $w_1 = 0.25, w_2 = 0.75$ and $\alpha = 0.01$.

```
#  0.250000 - 0.010000 * 1.000000( 0.750000 - 1.000000 )  => 0.252472
```

It gives the expected output.

In this part, I will implement the following formula to calculate the new bias

$$b' = b - \alpha(a - y)$$

Where α is the learning rate.

Procedure:

1. Setting the sign bit of α and y to 1 for the subtraction.
2. Compute $a - y$
3. Compute $-\alpha(a - y)$
4. Compute $b' = b - \alpha(a - y)$

Code:

```
prepared <= 1'b1;
b <= b_i;
a <= a_i;
y[31] <= 1;
y[30:0] <= y_i[30:0];
alpha[31] <= 1;
alpha[30:0] <= alpha_i[30:0];

#50
add1clk = 1;
#100;
multclk = 1;
#50;
add2clk = 1;
#50;
done = 1;
```

Explanation:

This code is similar to the code weighting function.

Testing & Results:

Due to the similarity of the code, I only present one test case.

Here we set the $w_1 = 0.25, w_2 = 0.75$ and $\alpha = 0.01$.

```
#  0.250000 - 0.010000 * ( 0.750000 - 1.000000 )  => 0.252472
```

It gives the expected output.

3.6 Loss function implementation

The loss function is basically $L(a, y) = -(y\log(a) + (1-y)\log(1-a))$. The log function is usually implemented using lookup table. We can also use the Taylor series approximation $\log(a) \approx (a-1) - \frac{(a-1)^2}{2} + \frac{(a-1)^3}{3}$.

Procedure of implementing the log function:

1. Compute $k = a - 1$
2. Compute k^2 and k^3
3. Compute $\frac{k^2}{2}$ and $\frac{k^3}{3}$
4. Compute $\log(a) \approx (a-1) - \frac{(a-1)^2}{2} + \frac{(a-1)^3}{3}$

Procedure of implementing the loss function:

1. Compute $1-a$ and $1-y$
2. Compute $\log(a)$ and $\log(1-a)$ parallelly
3. Compute $y\log(a)$ and $(1-y)\log(1-a)$ parallelly
4. Compute $(y\log(a) + (1-y)\log(1-a))$
5. Flip the 31st bit to get $L(a, y) = -(y\log(a) + (1-y)\log(1-a))$

This part is not difficult, just some arithmetic similar to above. However, I could not finish it due to the time limit.

4. Bonus - Investigation on a 3rd-party layout EDA tool, such as Magic VLSI

Background:

Magic VLSI is an open source software that maintained by universities and small companies, allowing VLSI engineers to implement their ideas in an easier way.

In this part, I implemented a CMOS inverter using Magic VLSI. I basically follow the tutorial in the following link.

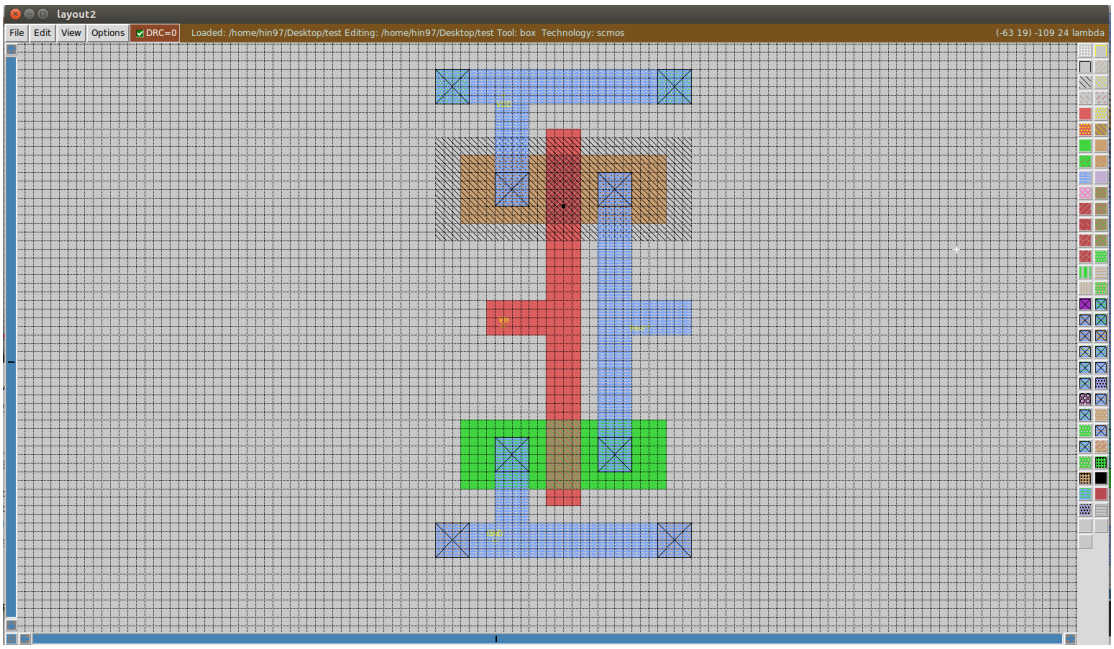
https://www.youtube.com/watch?v=52ZR7jrOuW0

Procedure:

1. Draw the well and diffusion layers
2. Draw the polysilicon layer
3. Draw the metal layers
4. Add connection

This tool is quite easy to use.

Layout:



5. Conclusion

This project allows me to gain more understanding on the computation of ANN, especially in the implementation of estimation and backward propagation.

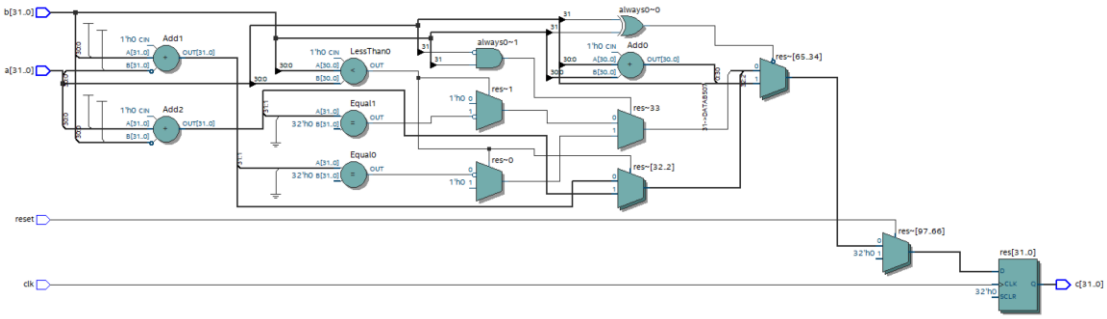
Also, I learned how to use Verilog. For example, designing modules and test bench, compilation of modules and debugging modules. Besides, I can apply the encapsulation technique in modules design, i.e. Combining modules together to form a larger module, hiding the information of sub-modules, which I think is a crucial concept in all engineering design problems.

Last but not least, I investigated a third-party EDA tool and draw layout using that. It allows me to understand the procedure in drawing layout and how to follow the design rules.

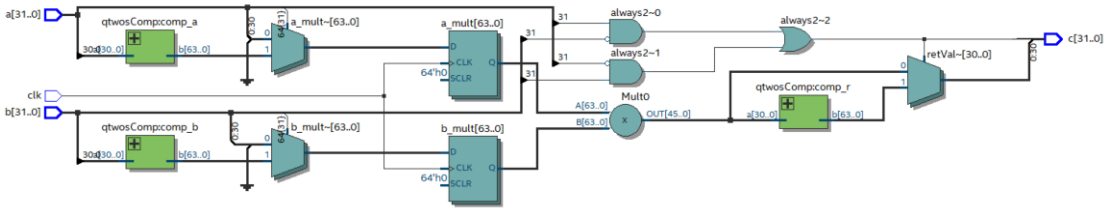
In conclusion, this project is quite difficult for me as I am new to the Verilog, but it was a fruitful experience to work on it.

Appendix – Gate Level RTL Viewer

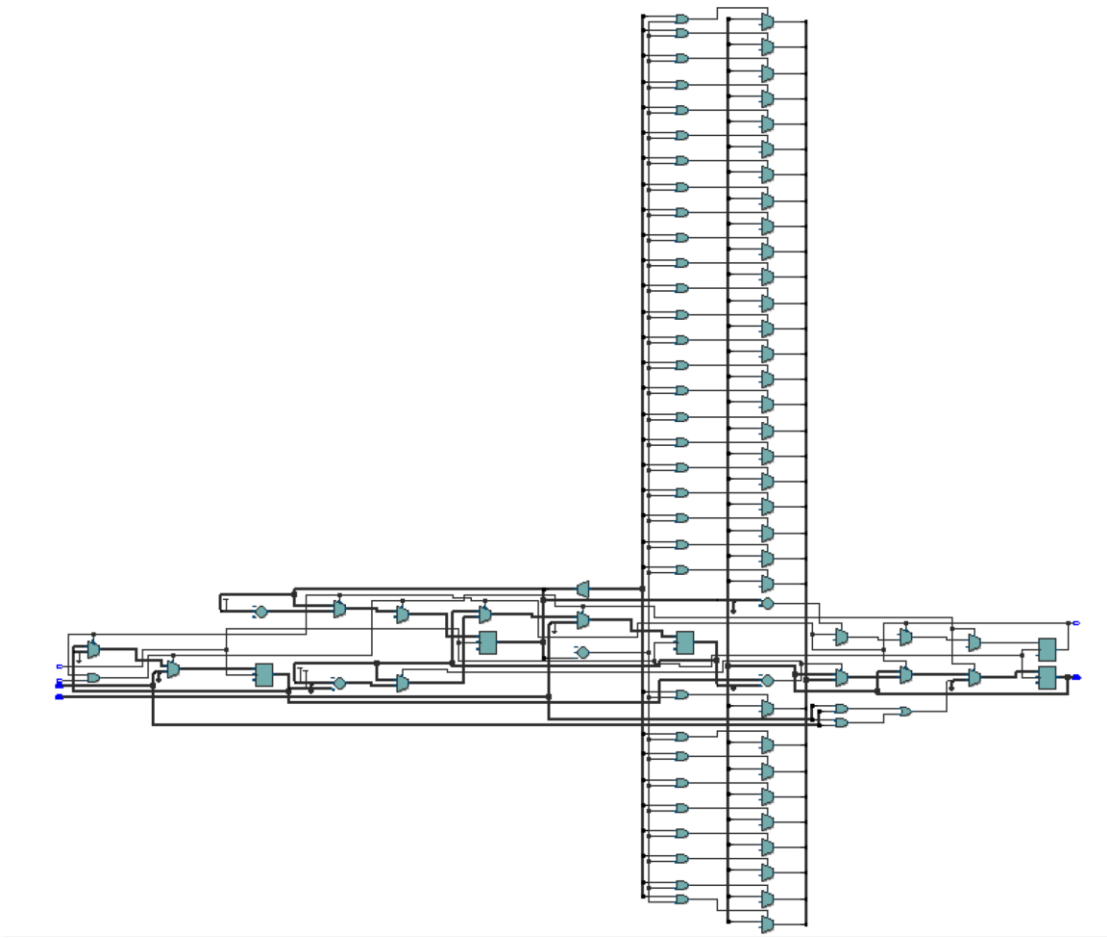
qadd



qmult

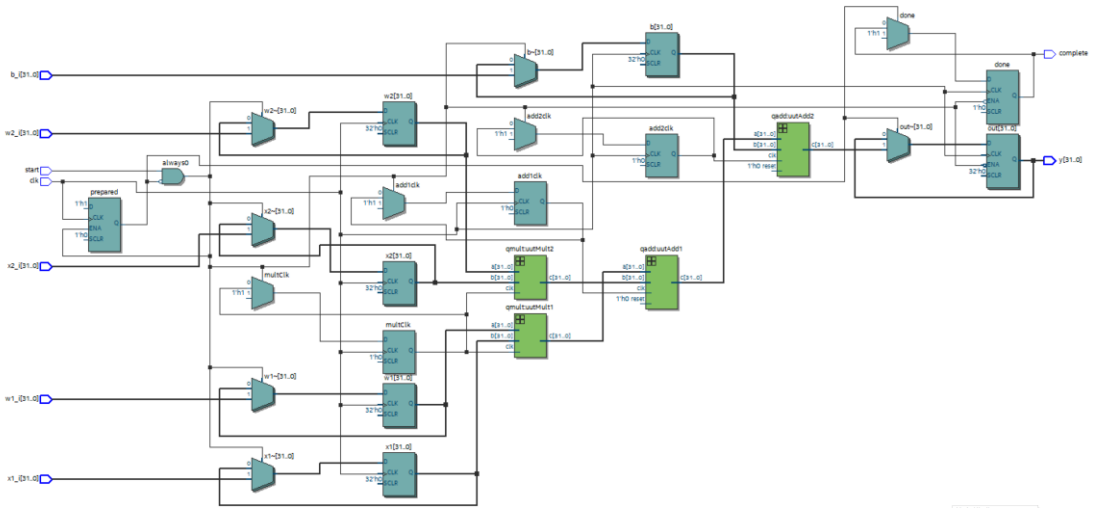


qdiv

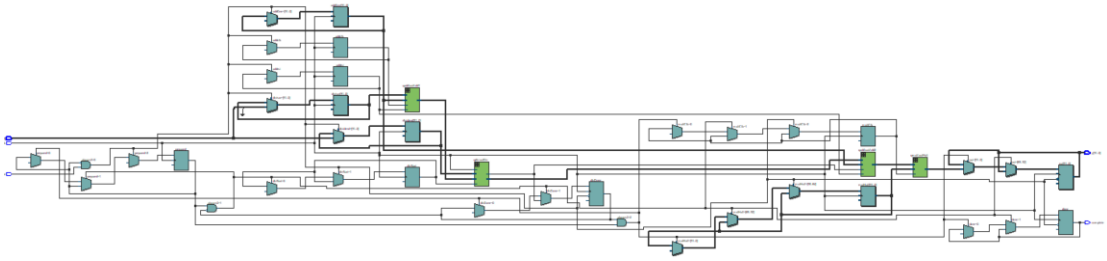


Forward Propagation

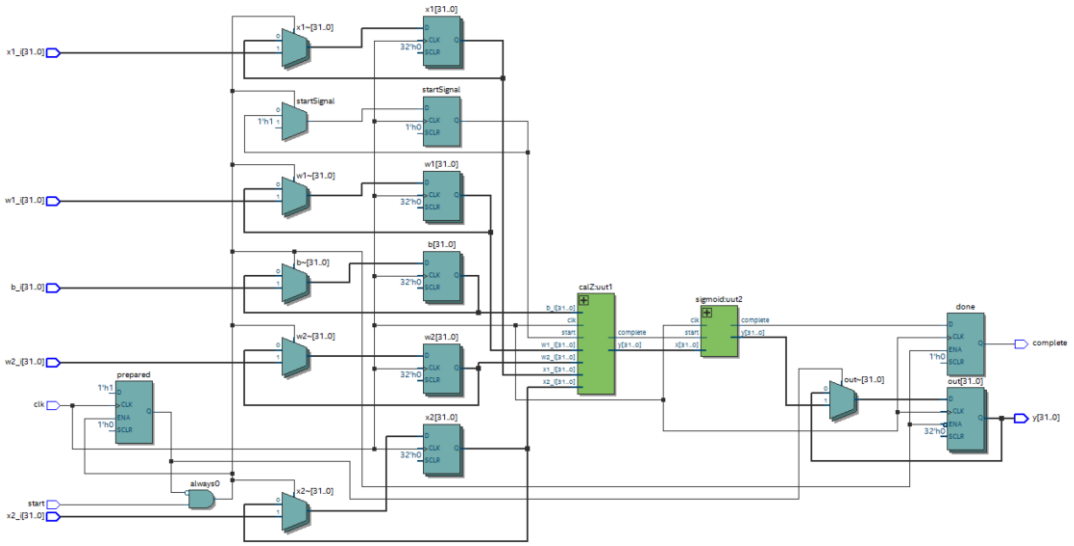
CalZ



Sigmoid

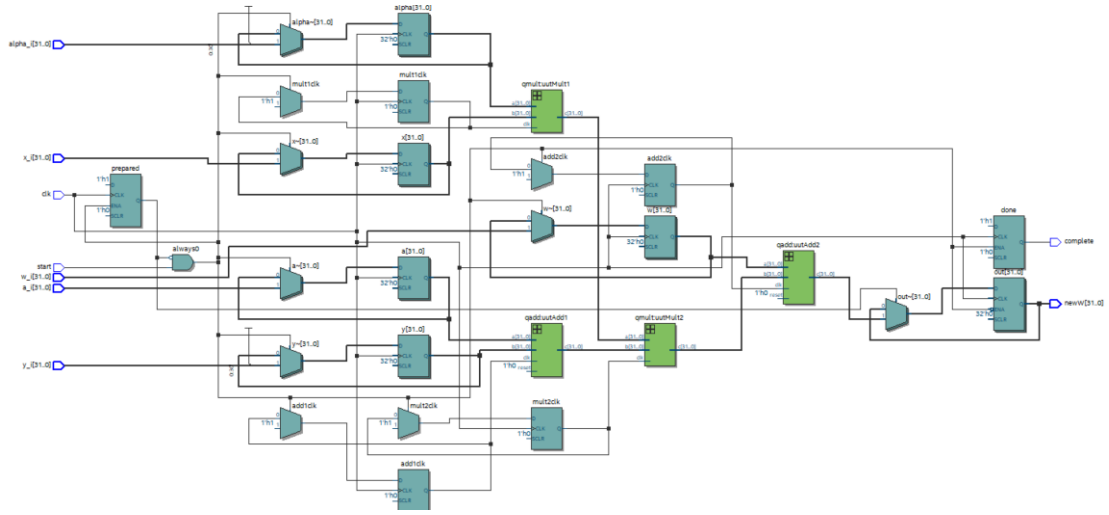


Forward



Backward Propagation

CalNewW



CalNewB