Weekly Report Zachary Ren

This week I finished verifying and testing a nearest one detector (NOD) design for an 8x8 multiplier in MATLAB. Using the proposed algorithm from the improved logarithmic multiplier and combining it with 4-bit leading one detector slices, I was able to develop a design that parallels existing leading one detectors. A 4-bit LOD slice is usually used in conjunction with other LODs for each respective "stage"; the leading one for each 4-bit section is evaluated, and another set of 4-bit LODs are used to compute the most significant out of all slices.

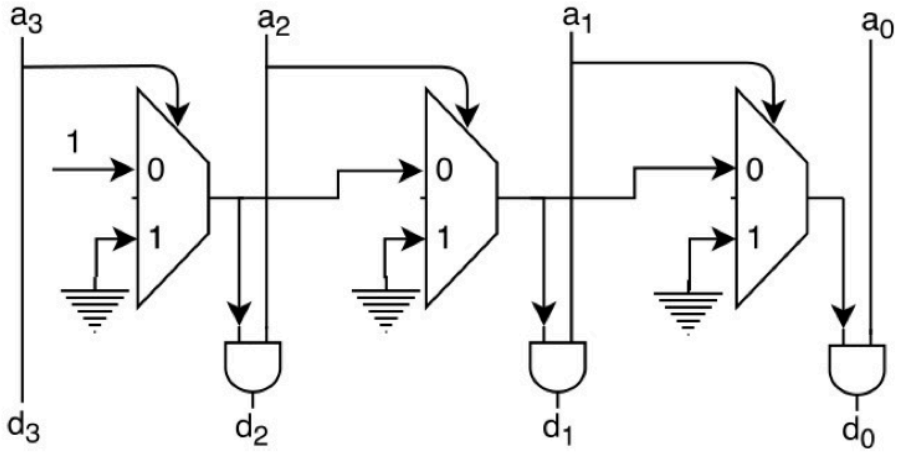


Figure 1. 4-bit Leading One Detector

LOD detectors are used for Mitchell approximation circuits. However, the Mitchell approximation method always results in an underestimation of the actual value. This is shown through the approximation used:

$$N = 2^k(1 + x)$$
$$log_2 N = k + log_2(1 + x) \approx k + x$$

Because the Mitchell method underestimates the value, it suffers from not having a double sided error distribution. The Improved logarithmic multiplier solves this issue.

Algorithm 1 Proposed approximation for $log_2 N$

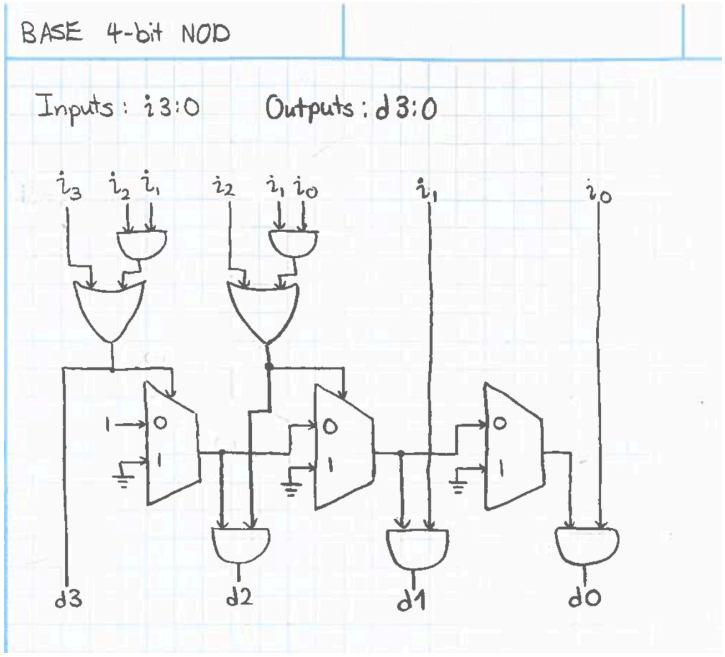
1: $N = 2^k(1 + x) = 2^{k+1}(1 - y)$
2: **if** $N - 2^k < 2^{(k+1)} - N$ **then** ▷ use underestimate
3: $x = N/2^k - 1$
4: $log_2 N \approx k + x$
5: **else** ▷ use overestimate
6: $y = 1 - N/2^{k+1}$
7: $log_2 N \approx k + 1 - y$
8: **end if**

The method requires using an extra bit during the multiplication step, as using sign-magnitude representation allows for an accurate computation of the final value, if the difference (q) between the actual value and the NOD is negative.

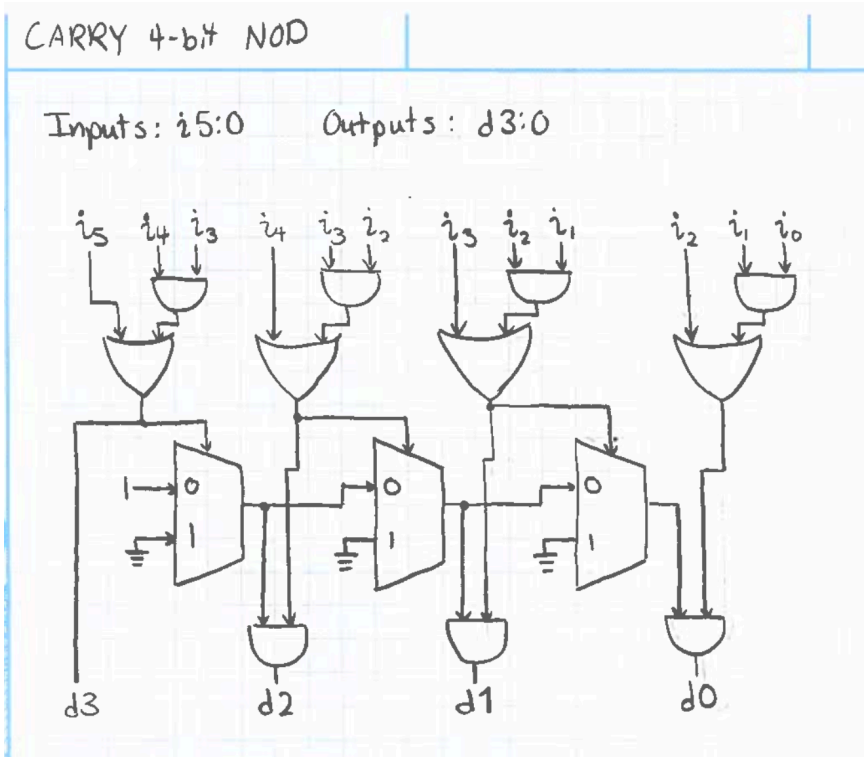
Algorithm 2 Proposed logarithmic multiplication

1: **procedure** M(A, B)
2: A, B: inputs, γ: approximate output
3: $2^{k_1} \leftarrow \text{NOD}(A)$,
4: $k_1 \leftarrow \text{PE}(2^{k_1})$,
5: $q_1 \leftarrow A - 2^{k_1}$, ▷ for steps 3-5 see (14)
6: $2^{k_2} \leftarrow \text{NOD}(B)$,
7: $k_2 \leftarrow \text{PE}(2^{k_2})$,
8: $q_2 \leftarrow B - 2^{k_2}$, ▷ for steps 6-8 see (15)
9: $q_1 2^{k_2} \leftarrow q_1 << k_2$,
10: $q_2 2^{k_1} \leftarrow q_2 << k_1$,
11: $2^{k_1+k_2} \leftarrow \text{DEC}(k_1 + k_2)$,
12: $\gamma \leftarrow 2^{k_1+k_2} + q_2 2^{k_1} + q_1 2^{k_2}$. ▷ see (16)

As for the Nearest One Detector itself, I have come up with a couple of designs. I like the divide and conquer aspect of most leading one detectors, being able to compute multiple operations at the same time, and I wanted to apply that same logic to the NOD. For a nearest one detector, the bit n is updated if both bit n-1 and bit n-2 are 1. Applying this logic, I was able to develop two NODs.

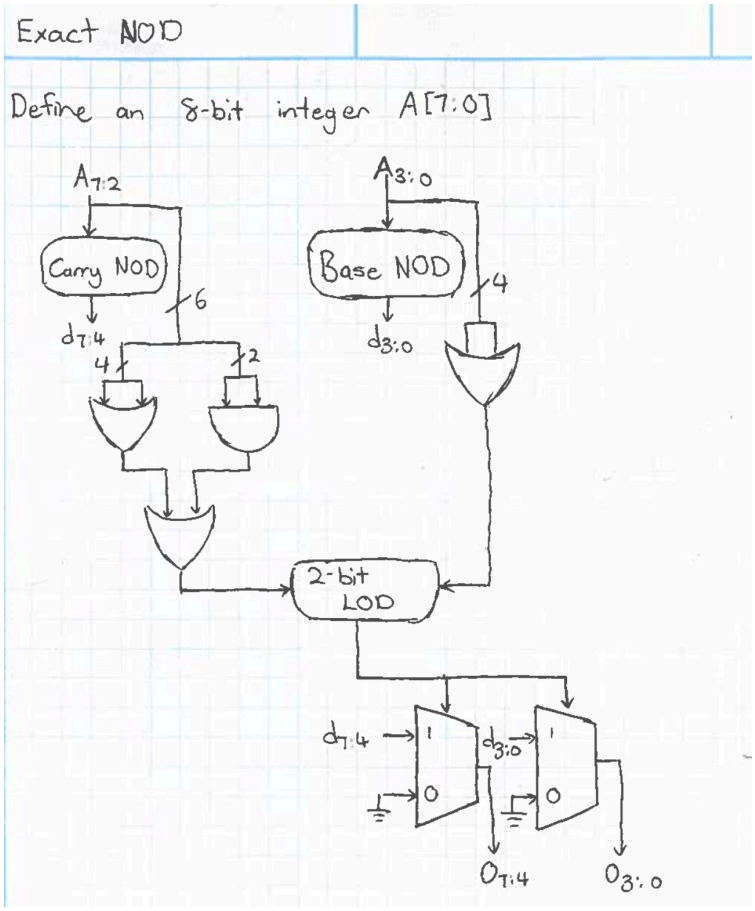


The first nearest one detector takes the least significant 4 inputs and processes the nearest one. This design is special, as it processes only the least significant 4-bits. Like a conventional LOD 4-bit slice, it uses a 2-to-1 multiplexer network to set the rest of the bits to 0 once a one is found. However, this NOD is missing a crucial component. If the bits d2 and d3 are equal to 1, d4 should be set to one. However, this bit is outside of the 4-bit range. For the rest of the 4-bit blocks the NOD needs to evaluate, an NOD with a carry is used:

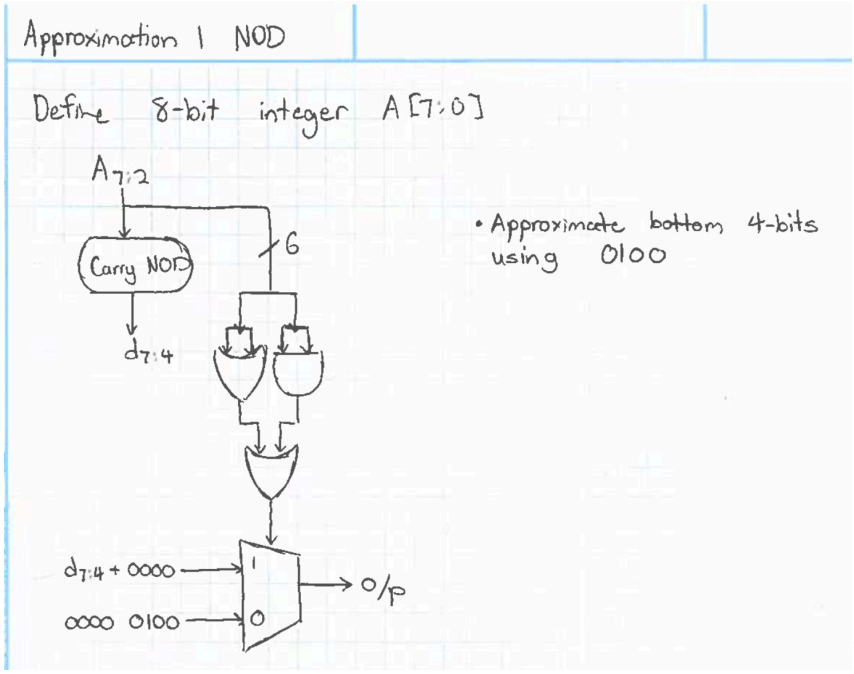


This nearest one detector takes the 6 bits as input; the 4 bits it is evaluating, and the 2 bits below that. In the case of an 8-bit multiplier; a carry NOD would evaluate bits 7:2 and a base NOD would evaluate bits 3:0. The carry NOD is able to take bits below the output bit range and use them to evaluate the two least significant bits. This way, no leading one is lost between different LOD slices.

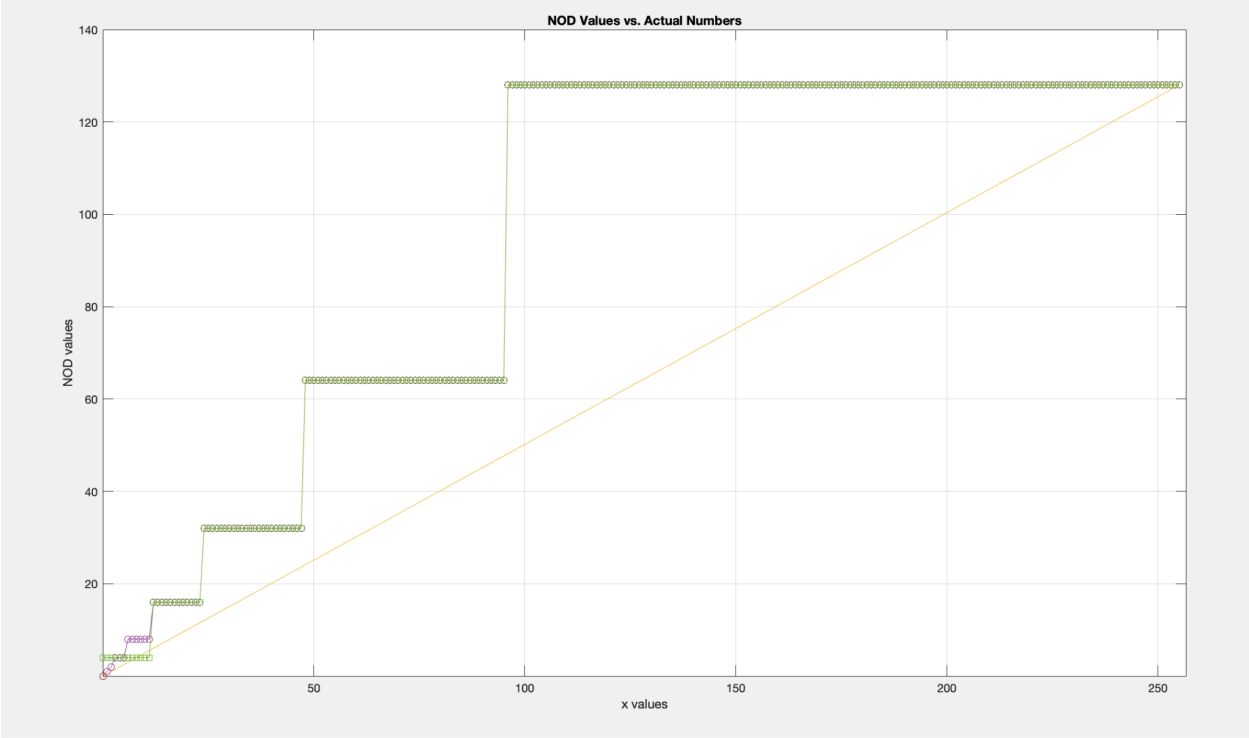
Exact 8-bit NOD:



Approximation 1 NOD: Sets least significant 4-bits to constant value



Using MATLAB, I have verified both circuits. Here is a visual representation of the decimal NOD value for the approximate and exact NOD plotted against the actual number.



The green line represents the Approximate LOD and the purple line represents the exact LOD. I have yet to calculate the error for both.

This week, I will start working on simulating the Improved Logarithmic Multiplier in Verilog. I will also take a look at some more literature for approximation strategies and testing.

Sources:

[1] M. S. Ansari, B. F. Cockburn and J. Han, "An Improved Logarithmic Multiplier for Energy-Efficient Neural Computing," in IEEE Transactions on Computers, vol. 70, no. 4, pp. 614-625, 1 April 2021, doi: 10.1109/TC.2020.2992113.
keywords: {Artificial neural networks;Hardware;Adders;Neurons;Biological neural networks;Training;Energy consumption;Neural network;logarithmic multiplier;adder;energy efficiency;error-tolerant},
[2] S. Gandhi, M. S. Ansari, B. F. Cockburn and J. Han, "Approximate Leading One Detector Design for a Hardware-Efficient Mitchell Multiplier," 2019 IEEE Canadian Conference of Electrical and Computer Engineering (CCECE), Edmonton, AB, Canada, 2019, pp. 1-4, doi: 10.1109/CCECE.2019.8861800.
keywords: {Adders;Hardware;Multiplexing;Detectors;Conferences;Simulation;Delays;leading one detector;approximate adder;Mitchell logarithmic multiplier;approximate arithmetic},

Matlab Code:

```
fprintf('\nStart of Program\n');
x = 0:255;
x_values = zeros(size(x));
NOD_values = zeros(size(x));
ANOD_values = zeros(size(x));
for i=1:1:256
   %fprintf("%d: ", i);
   %fprintf(dec2bin(i,8));
   %fprintf(">>>")
   NOD_values(i) = NOD(i-1);
   ANOD_values(i) = ApproxNOD(i-1);
   x_values(i) = i-1;
   fprintf("\n");
end
plot(x_values, NOD_values, 'o-');
hold on;
plot(x_values, ANOD_values, 's-');
xlabel('x values');
ylabel('NOD values');
title('NOD Values vs. Actual Numbers');
grid on;
%Exact
function decimalNOD = NOD(binaryNumber)
   bitnod = zeros(1,8);
   bitValue = zeros(1,8);
   for k=8:-1:1
       bitValue(9-k) = bitget(binaryNumber,k);
   end
   bit7to4OR = or(or(bitValue(1),bitValue(2)),or(bitValue(3),bitValue(4)));
   bit3to2AND = and(bitValue(5), bitValue(6));
   bit7to4ip = or(bit3to2AND, bit7to4OR);
   bit3to0OR = or(or(bitValue(5),bitValue(6)),or(bitValue(7),bitValue(8)));
   outputvar = twobitLOD(bit7to4ip, bit3to0OR);
   bitnod(1:4) = set2to1mux(0000, CarryNOD(binaryNumber), outputvar(1));
   bitnod(5:8) = set2to1mux(0000, BaseNOD(binaryNumber), outputvar(2));
  % fprintf("%d%d%d%d%d%d%d%d", bitnod(1), bitnod(2), bitnod(3), bitnod(4),
bitnod(5), bitnod(6), bitnod(7), bitnod(8));
   binaryString = num2str(bitnod);
   decimalNOD = bin2dec(binaryString);
end
function decimalANOD = ApproxNOD(binaryNumber)
   bitanod = zeros(1,8);
   bitValue = zeros(1,8);
   bits4zero = zeros(1,4);
   bits4zero(1) = 0;
   bits4zero(2) = 1;
   bits4zero(3) = 0;
```

```
    bits4zero(4) = 0;
    for k=8:-1:1
        bitValue(9-k) = bitget(binaryNumber,k);
    end
    bit7to4OR = or(or(bitValue(1),bitValue(2)),or(bitValue(3),bitValue(4)));
    bit3to2AND = and(bitValue(5), bitValue(6));
    bit7to4ip = or(bit3to2AND, bit7to4OR);
    bitanod(1:4) = set2to1mux(0000, CarryNOD(binaryNumber), bit7to4ip);
    bitanod(5:8) = set2to1mux(bits4zero, 0000, bit7to4ip);
    %fprintf("%d%d%d%d%d%d%d%d", bitanod(1), bitanod(2), bitanod(3), bitanod(4),
bitanod(5), bitanod(6), bitanod(7), bitanod(8));
    binaryString = num2str(bitanod);
    decimalANOD = bin2dec(binaryString);
end
%top 4 bits
function bits7to4 = CarryNOD(input1)
    bitValue = zeros(1,6);
    bits7to4 = zeros(1,4);
    for k=8:-1:3
        bitValue(9-k) = bitget(input1,k);
    end
    combbit1 = or(bitValue(1), and(bitValue(2),bitValue(3)));
    combbit2 = or(bitValue(2), and(bitValue(3),bitValue(4)));
    combbit3 = or(bitValue(3), and(bitValue(4),bitValue(5)));
    combbit4 = or(bitValue(4), and(bitValue(5),bitValue(6)));
    ctrl1 = set2to1mux(1, 0, combbit1);
    ctrl2 = set2to1mux(ctrl1, 0, combbit2);
    ctrl3 = set2to1mux(ctrl2, 0, combbit3);
    bits7to4(1) = combbit1;
    bits7to4(2) = and(combbit2, ctrl1);
    bits7to4(3) = and(combbit3, ctrl2);
    bits7to4(4) = and(combbit4, ctrl3);
    %fprintf('Bits 7 to 4: ');
    %fprintf('%d', bitValue);
    %fprintf('\n');
    %fprintf('%d', bits7to4);
    %fprintf('\n');
end
%bottom 4 bits
function bits3to0 = BaseNOD(input1)
    %fprintf('Bits 3 to 0: ');
    bitValue = zeros(1,4);
    bits3to0 = zeros(1,4);
    for k=4:-1:1
        bitValue(5-k) = bitget(input1,k);
    end
    combbit1 = or(bitValue(1), and(bitValue(2),bitValue(3)));
    combbit2 = or(bitValue(2), and(bitValue(3),bitValue(4)));
```

```
    ctrl1 = set2to1mux(1, 0, combbit1);
    ctrl2 = set2to1mux(ctrl1, 0, combbit2);
    ctrl3 = set2to1mux(ctrl2, 0, bitValue(3));
    bits3to0(1) = combbit1;
    bits3to0(2) = and(combbit2, ctrl1);
    bits3to0(3) = and(bitValue(3), ctrl2);
    bits3to0(4) = and(bitValue(4), ctrl3);
    %fprintf('%d', bitValue);
    %fprintf('\n');
    %fprintf('%d', bits7to4);
    %fprintf('\n');
end
%control bits - 2bit LOD
function bitsz1to0 = twobitLOD(input1, input2)
    bitsz1to0 = zeros(1,2);
    ctrl1 = set2to1mux(1, 0, input1);
    bitsz1to0(1) = input1;
    bitsz1to0(2) = and(input2, ctrl1);
    %fprintf('%d', bitsz1to0);
end
%mux 2to1
function output = set2to1mux(D_0, D_1, select_line)
    if(select_line==0)
        output = D_0;
    else
        output = D_1;
    end
end
```