XNOR-Net and Binary Connect Hardware Design

Lucas B., Bibrak C., Adam D., Prateek S.

E501 Design Project Presentation

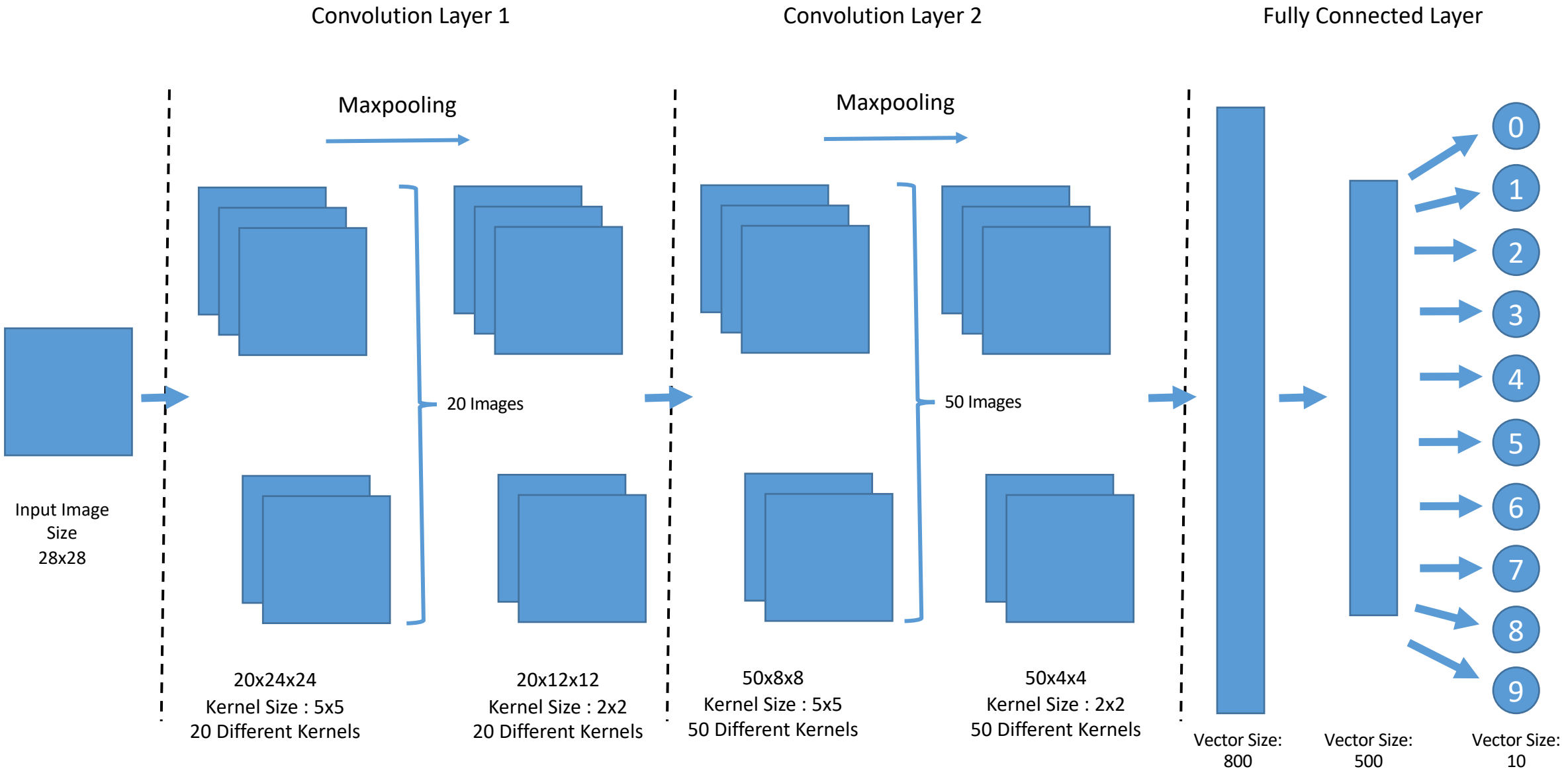
4/16/2018

Introduction

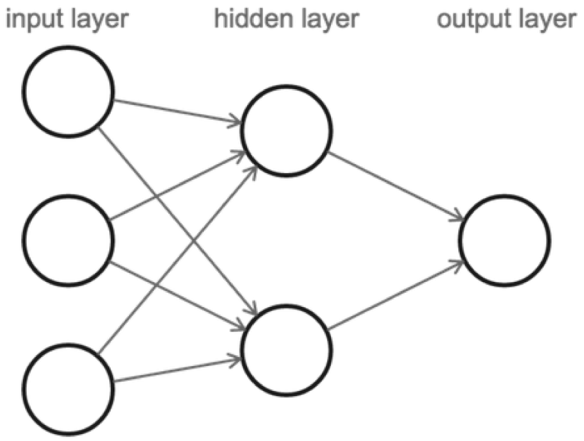
- Hardware designs were created to implement the basic hardware building blocks of the following papers:
  - “XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks”
  - “BinaryConnect: Training Deep Neural Networks with binary weights during propagations”

Feed-forward network using a trained neural network model

- For the current scope of the project, we consider MNIST dataset which consists of 10 classes of images i.e. 0 to 9.
- Once the training of the neural network is done using any machine, our hardware design focuses on achieving speedup in the classification phase.
- The weights learned by the neural network will be inserted in the module using a BRAM to the feed forward network.
- These weights along with the images to be classified would be used as an input to the feed forward network on an FPGA.
- The scope of our project is only defined for the implementation of the fully connected layer after the convolutions have been done.


Convolution Layer 1
Convolution Layer 2
Fully Connected Layer
Maxpooling
Maxpooling
20 Images
50 Images
Input Image
Size
28x28
20x24x24
Kernel Size : 5x5
20 Different Kernels
20x12x12
Kernel Size : 2x2
20 Different Kernels
50x8x8
Kernel Size : 5x5
50 Different Kernels
50x4x4
Kernel Size : 2x2
50 Different Kernels
Vector Size:
800
Vector Size:
500
Vector Size:
10
0
1
2
3
4
5
6
7
8
9

Fully Connected Layer

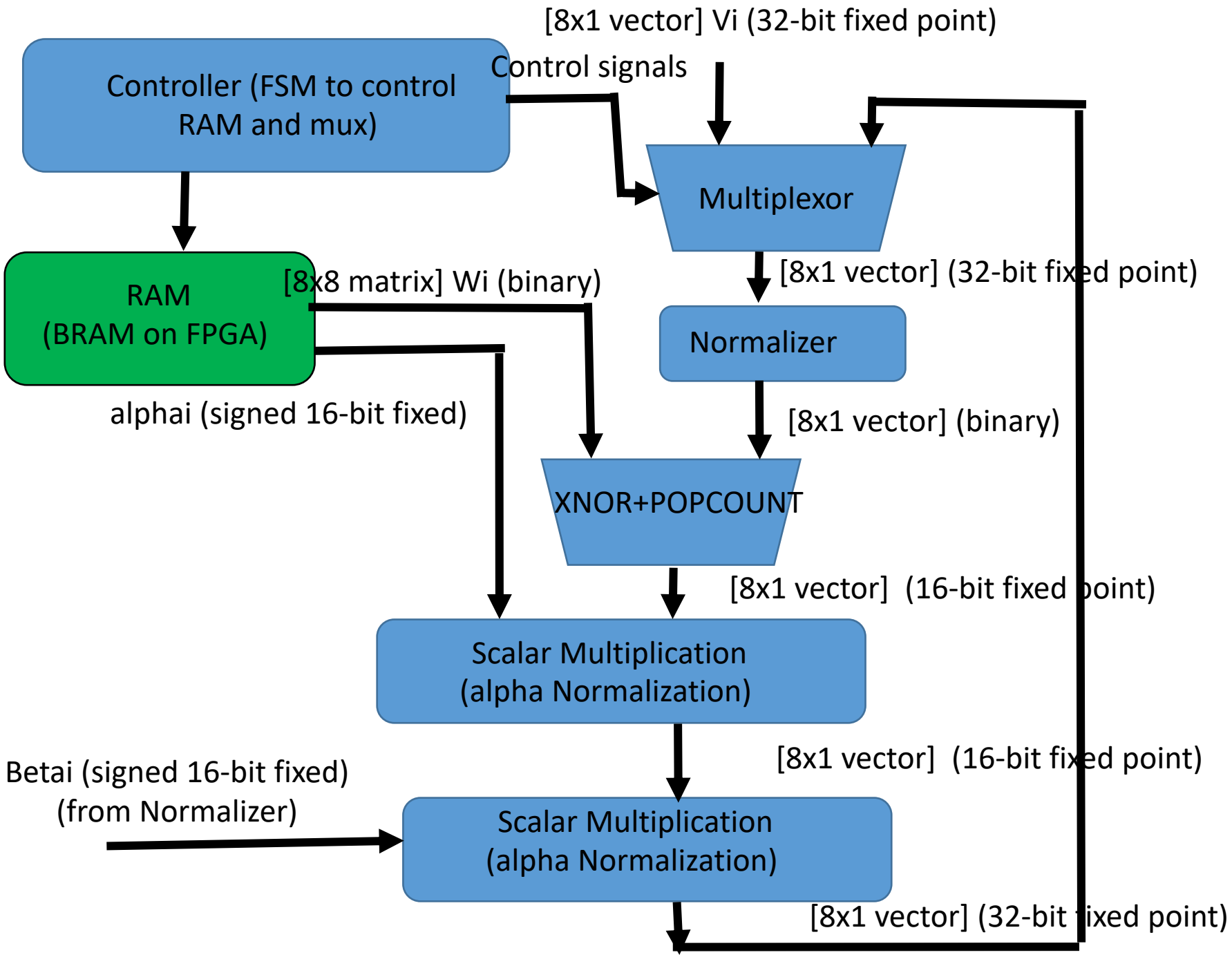


Input Layer becomes a vector, the weights which are multiplied and accumulated become a matrix. This whole process of Matrix and Vector multiplication along with accumulation is simply a Matrix-Vector Multiplication.

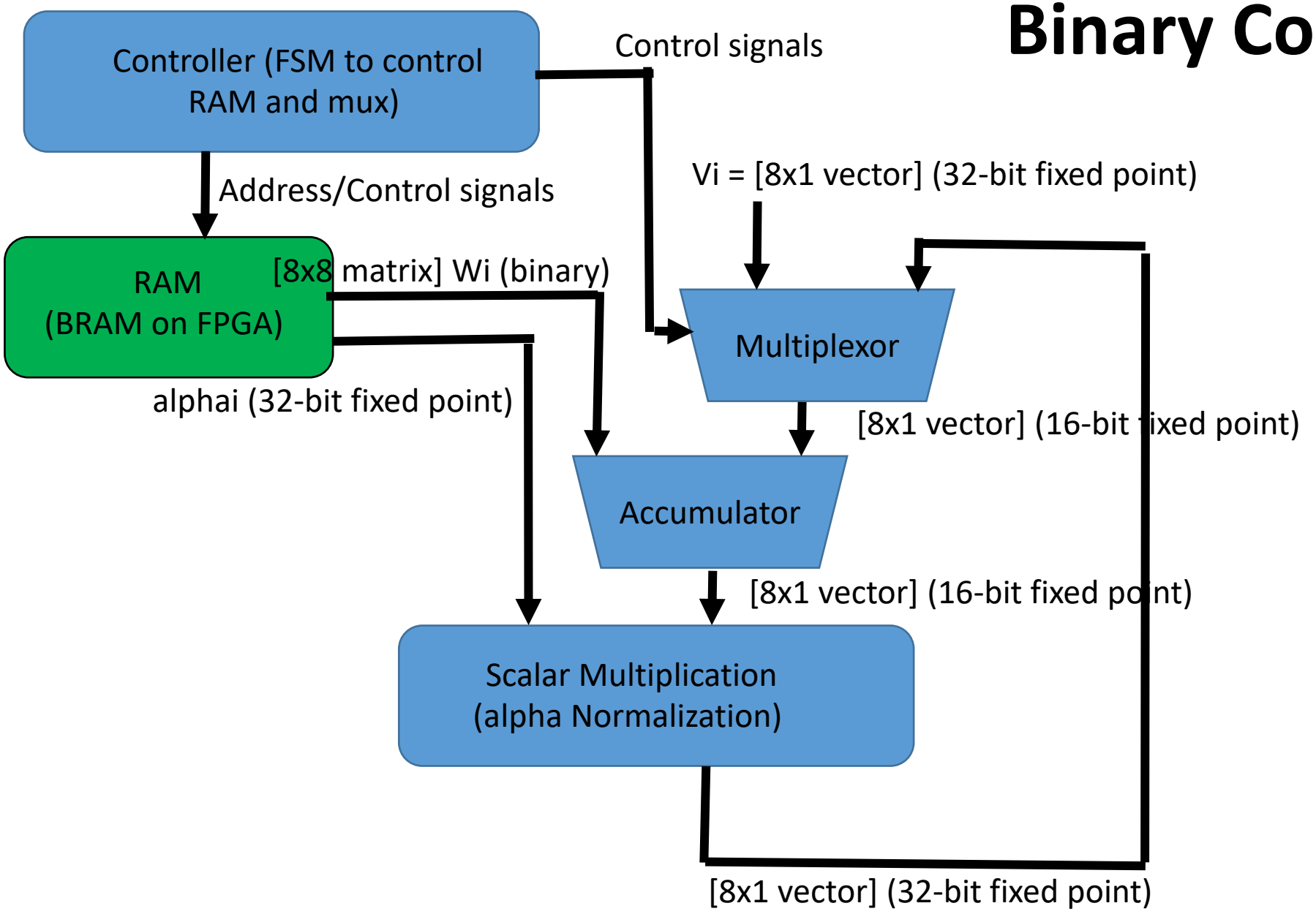
Input Vector (I) : 800 x 1
Weight Matrix (W1) : 500 x 800
Hidden Layer (H) : H = W1 x I = [500 x 800] [800 x 1] = [500 x 1]
Weight Matrix (W2) : 10 x 500
Output Layer (O) : O = W2 x H = [10 x 500] [500 x 1] = [10 x 1]

This output layer which is a vector of 10 values classifies all the images in the possible 10 numbers according to MNIST Data Set we use.

XNOR-Net


[8x1 vector] Vi (32-bit fixed point)
Control signals
Controller (FSM to control RAM and mux)
Multiplexor
[8x1 vector] (32-bit fixed point)
RAM (BRAM on FPGA)
[8x8 matrix] Wi (binary)
Normalizer
alphai (signed 16-bit fixed)
[8x1 vector] (binary)
XNOR+POPCOUNT
[8x1 vector] (16-bit fixed point)
Scalar Multiplication (alpha Normalization)
[8x1 vector] (16-bit fixed point)
Betai (signed 16-bit fixed) (from Normalizer)
Scalar Multiplication (alpha Normalization)
[8x1 vector] (32-bit fixed point)

Binary Connect

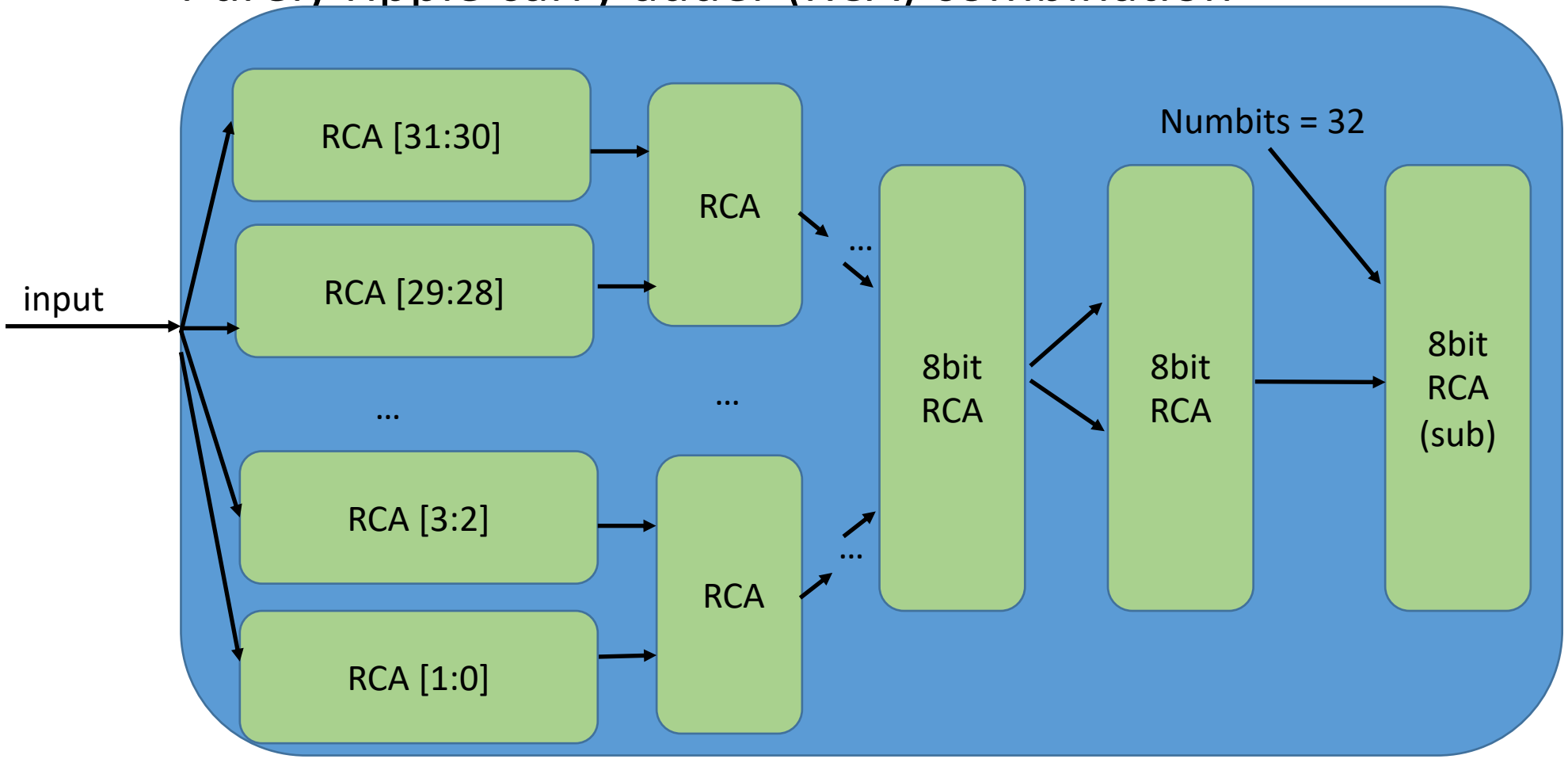


XNOR-Net Core Components	Binary Connect Core Components
XNOR + Popcount Design Concept Performs multiplication and accumulation(MAC) operation optimizations.	**Accumulator** Performs multiplication and accumulation(MAC) operation optimizations.
Alpha and Beta Multipliers Performs multiplication of vector to scalar	**Alpha Multiplier** Performs multiplication of vector to scalar

Modules

Popcount Design Concept 1 (Slow)

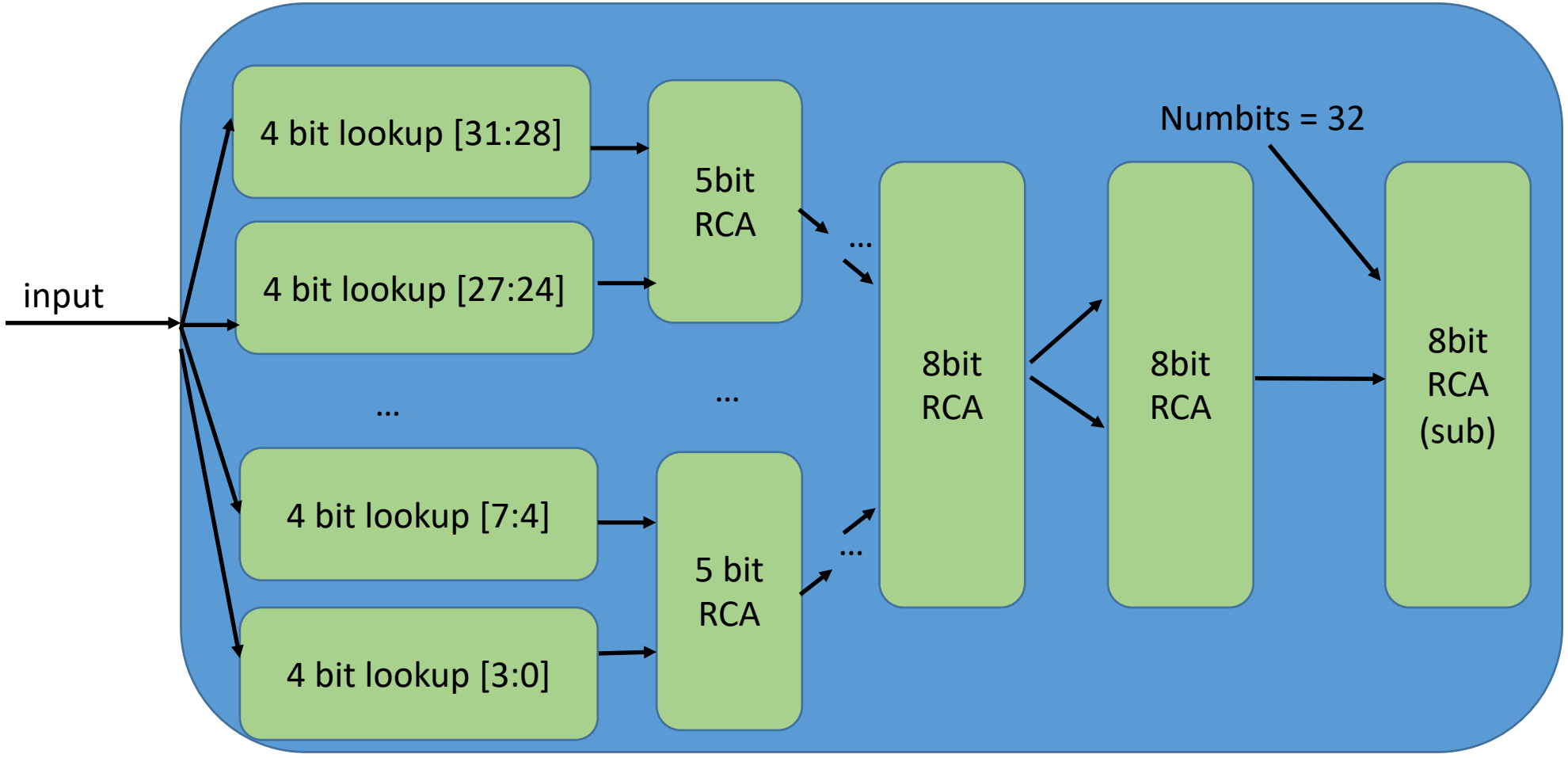
- Purely ripple carry adder (RCA) combination



- Also, If you simply count the number of ones in a string, you need to apply:
  - POPCOUNT = (NUMBER OF ONES) * 2 – (NUMBER OF BITS)

Popcount Design Concept 2 (Better)

- 8 of 4-bit Lookups + ripple carry adder (RCA) combination
- Look up table saves adders and reduces propagation time



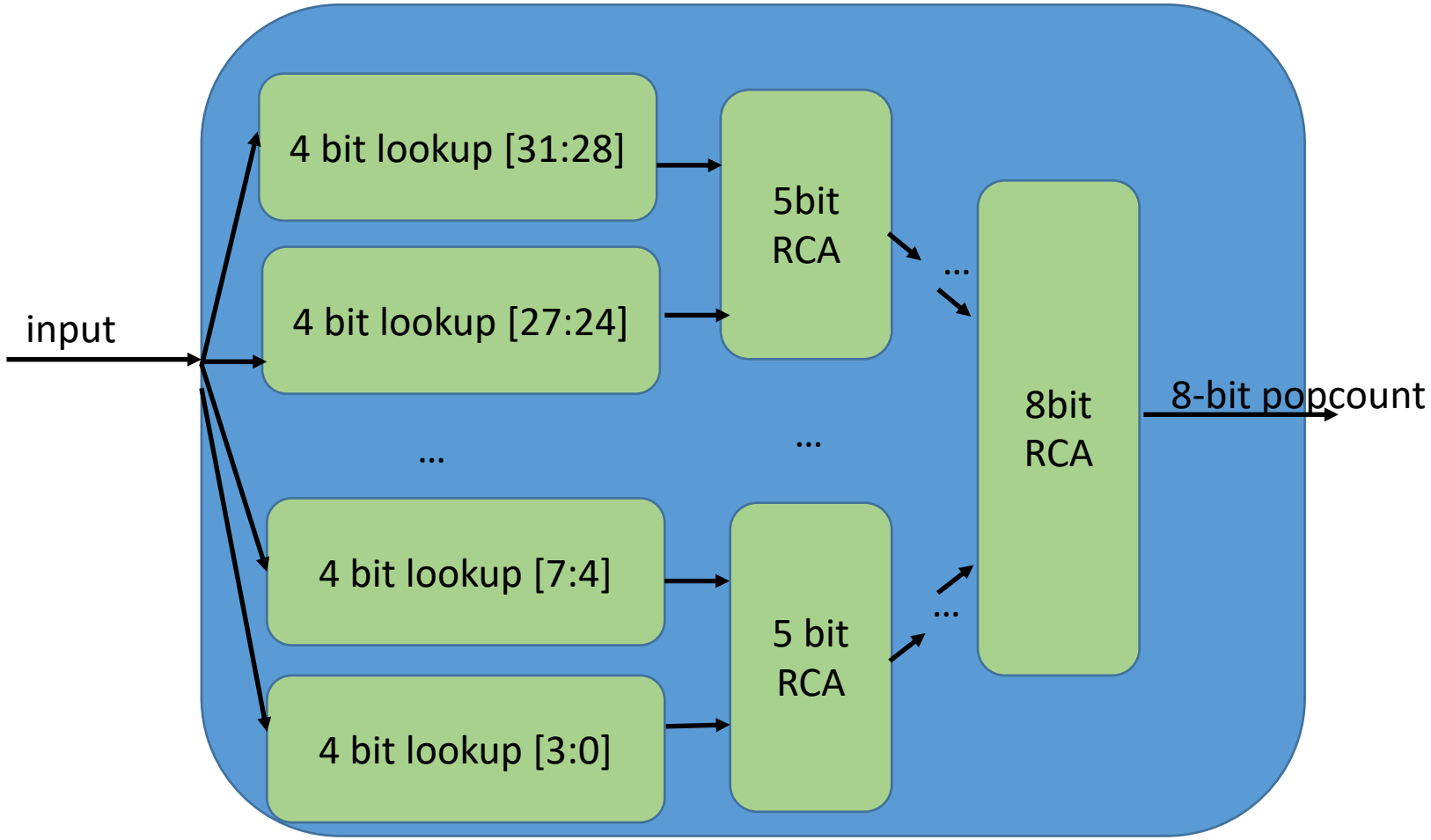
- Still, you simply count the number of ones in a string, you need to apply:
  - POPCOUNT = (NUMBER OF ONES) * 2 – (NUMBER OF BITS)

4 bit lookup table

Input [3:0]	Popcount
0000	0
0001	1
0010	1
0011	2
0100	1
0101	2
0110	2
0111	3
1000	1
1001	2
1010	2
1011	3
1100	2
1101	3
1110	3
1111	4

Popcount Design Concept 3 (Best)

- 8 of 4-bit Lookups + ripple carry adder (RCA) combination



- Building the negative popcount values into LUT:
  - Saves 2 RCA components

4 bit lookup

Input [3:0]	Popcount
0000	-4
0001	-2
0010	-2
0011	0
0100	-2
0101	0
0110	0
0111	2
1000	-2
1001	0
1010	0
1011	2
1100	0
1101	2
1110	2
1111	4

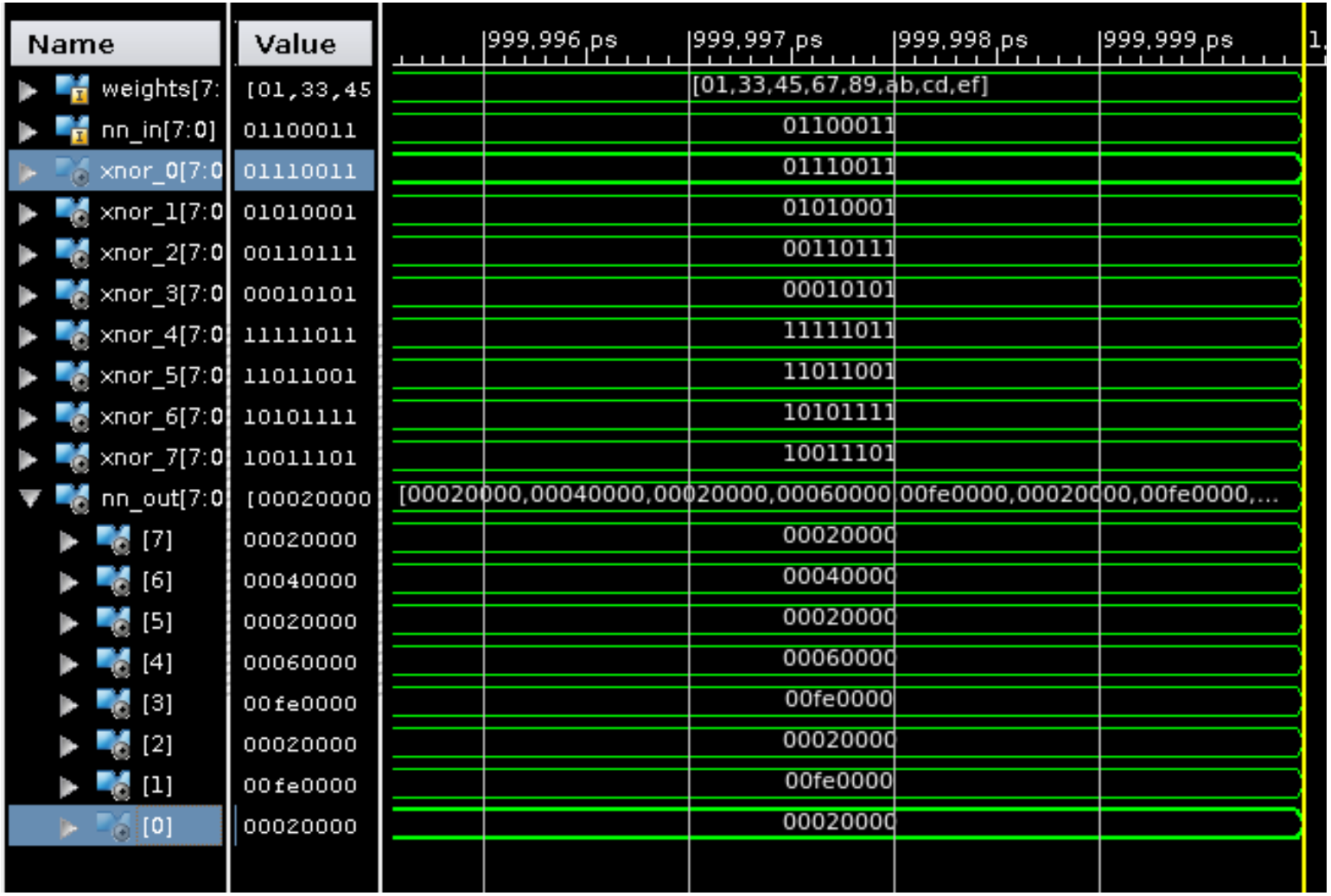
8-bit Popcount Design

```
entity Popcount_4 is
    Port ( input_vector : in  STD_LOGIC_VECTOR(3 downto 0);
           count : out  STD_LOGIC_VECTOR(7 downto 0)
           );
end Popcount_4;

architecture Behavioral of Popcount_4 is

begin
  process (input_vector)
  begin
  -- 2s comp notation for 8 bits
  -- -4 = "FC"
  -- -2 = "FE"
  -- 0 = x"00"
  -- +2 = x"02"
  -- +4 = x"04"

    case input_vector is
       when x"0"  => count <= x"FC";
       when x"1" | x"2" | x"4" | x"8" => count <= x"FE";
       when x"3" | x"5" | x"6" | x"9" | x"A" | x"C" => count <= x"00";
       when x"7" | x"B" | x"D" | x"E" => count <= x"02";
       when x"F"  => count <= x"04";
       when others => count <= x"00";
    end case;
  end process;
end Behavioral;
```

8-bit Style	# RCA	# LUT4	Speed	Area
Just RCA	9	0	slow	large
RCA+LUT	3	2	fast	med
RCA+LUT opt	1	2	fast	small

```
entity Popcount_8 is
    Port ( input_vector : in  STD_LOGIC_VECTOR(31 downto 0);
           count : out  STD_LOGIC_VECTOR(7 downto 0)
           );
end Popcount_8;

architecture Behavioral of Popcount_8 is

component Popcount_4
   port ( input_vector : in  STD_LOGIC_VECTOR(3 downto 0);
           count : out  STD_LOGIC_VECTOR(7 downto 0)
           );
   end component;

component Ripple_Carry_Adder
    Port ( SUB: in STD_LOGIC; -- '0' = add, '1' = sub
           A : in  STD_LOGIC_VECTOR (7 downto 0);
           B : in  STD_LOGIC_VECTOR (7 downto 0);
           S : out  STD_LOGIC_VECTOR (7 downto 0);
           COUT : out  STD_LOGIC;
           V : out STD_LOGIC
           );
   end component;

  signal count1 : std_logic_vector(7 downto 0) := x"00";
  signal count0 : std_logic_vector(7 downto 0) := x"00";

begin
  pop1: Popcount_4 PORT MAP (input_vector => input_vector(23 downto 20), count => count1);
  pop0: Popcount_4 PORT MAP (input_vector => input_vector(19 downto 16), count => count0);
  add_0_0: Ripple_Carry_Adder PORT MAP (SUB => '0', A => count1, B => count0, S => count);

end Behavioral;
```

XNOR and Popcount Results



Scalar Multipliers

Multiplies a fixed point scalar value to an integer

```
 8
 9  entity scalar_multiplier_ard is
10      Port ( scalar : in  sfixed (7 downto -8);
11             nn_in : in vector_fixed_point_type;
12             nn_out : out sfixed_fixed_point_type
13             );
14  end scalar_multiplier_ard;
15
16  architecture Behavioral of scalar_multiplier_ard is
17
18  begin
19
20    nn_out(0) <= scalar * to_sfixed(to_integer(unsigned(nn_in(0)(31 downto 16))),7,-8);
21    nn_out(1) <= scalar * to_sfixed(to_integer(unsigned(nn_in(1)(31 downto 16))),7,-8);
22    nn_out(2) <= scalar * to_sfixed(to_integer(unsigned(nn_in(2)(31 downto 16))),7,-8);
23    nn_out(3) <= scalar * to_sfixed(to_integer(unsigned(nn_in(3)(31 downto 16))),7,-8);
24    nn_out(4) <= scalar * to_sfixed(to_integer(unsigned(nn_in(4)(31 downto 16))),7,-8);
25    nn_out(5) <= scalar * to_sfixed(to_integer(unsigned(nn_in(5)(31 downto 16))),7,-8);
26    nn_out(6) <= scalar * to_sfixed(to_integer(unsigned(nn_in(6)(31 downto 16))),7,-8);
27    nn_out(7) <= scalar * to_sfixed(to_integer(unsigned(nn_in(7)(31 downto 16))),7,-8);
28
29  end Behavioral;
30
```

Scalar Multipliers

Multiplies a fixed point scalar value to a fixed point

```
 9  entity scalar_multiplier_last is
10      Port ( scalar : in  sfixed (7 downto -8);
11             nn_in : in sfixed_fixed_point_type;
12             nn_out : out sfixed_fixed_point_type
13             );
14  end scalar_multiplier_last;
15
16  architecture Behavioral of scalar_multiplier_last is
17
18  begin
19
20    nn_out(0) <= scalar * nn_in(0)(7 downto -8);
21    nn_out(1) <= scalar * nn_in(1)(7 downto -8);
22    nn_out(2) <= scalar * nn_in(2)(7 downto -8);
23    nn_out(3) <= scalar * nn_in(3)(7 downto -8);
24    nn_out(4) <= scalar * nn_in(4)(7 downto -8);
25    nn_out(5) <= scalar * nn_in(5)(7 downto -8);
26    nn_out(6) <= scalar * nn_in(6)(7 downto -8);
27    nn_out(7) <= scalar * nn_in(7)(7 downto -8);
28
29  end Behavioral;
30
31
```

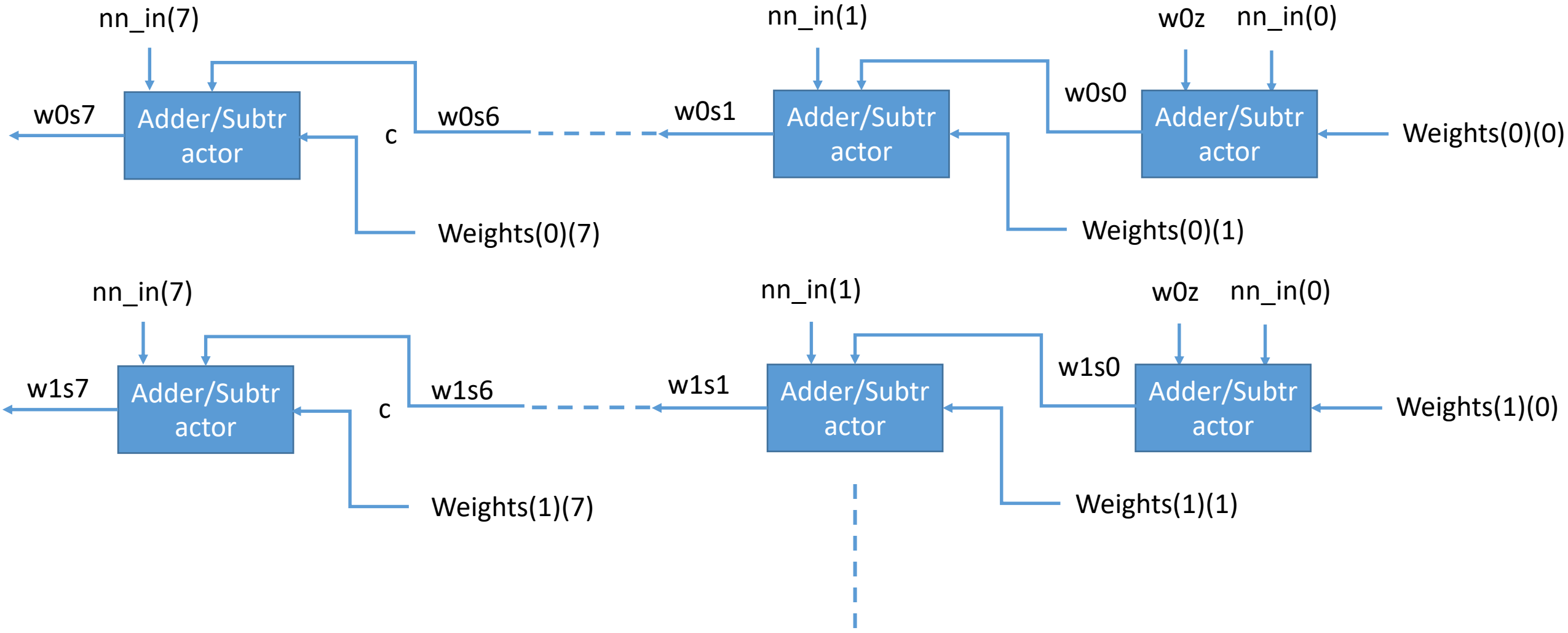
Accumulator

- This module is the core of Binary Connect.
- Multiplying the fixed point numbers to the binary weight matrix essentially is not multiplication. It is addition or subtraction depending on the corresponding weight in the weight matrix. For example, a three dimensional binary weight matrix and an input vector when multiplied behaves as follows:

$$\begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 1.0 \\ 2.5 \\ 3.5 \end{bmatrix} = \begin{bmatrix} 1 - 2.5 + 3.5 \\ 1 + 2.5 - 3.5 \\ -1 - 2.5 - 3.5 \end{bmatrix}$$

- Accumulator thus, removes the multiplication and replaces it with addition and subtraction operations.
- As the accumulator is cheaper than the multiplier, its usage in the neural network design grants speedup.

Block Diagram for accumulator



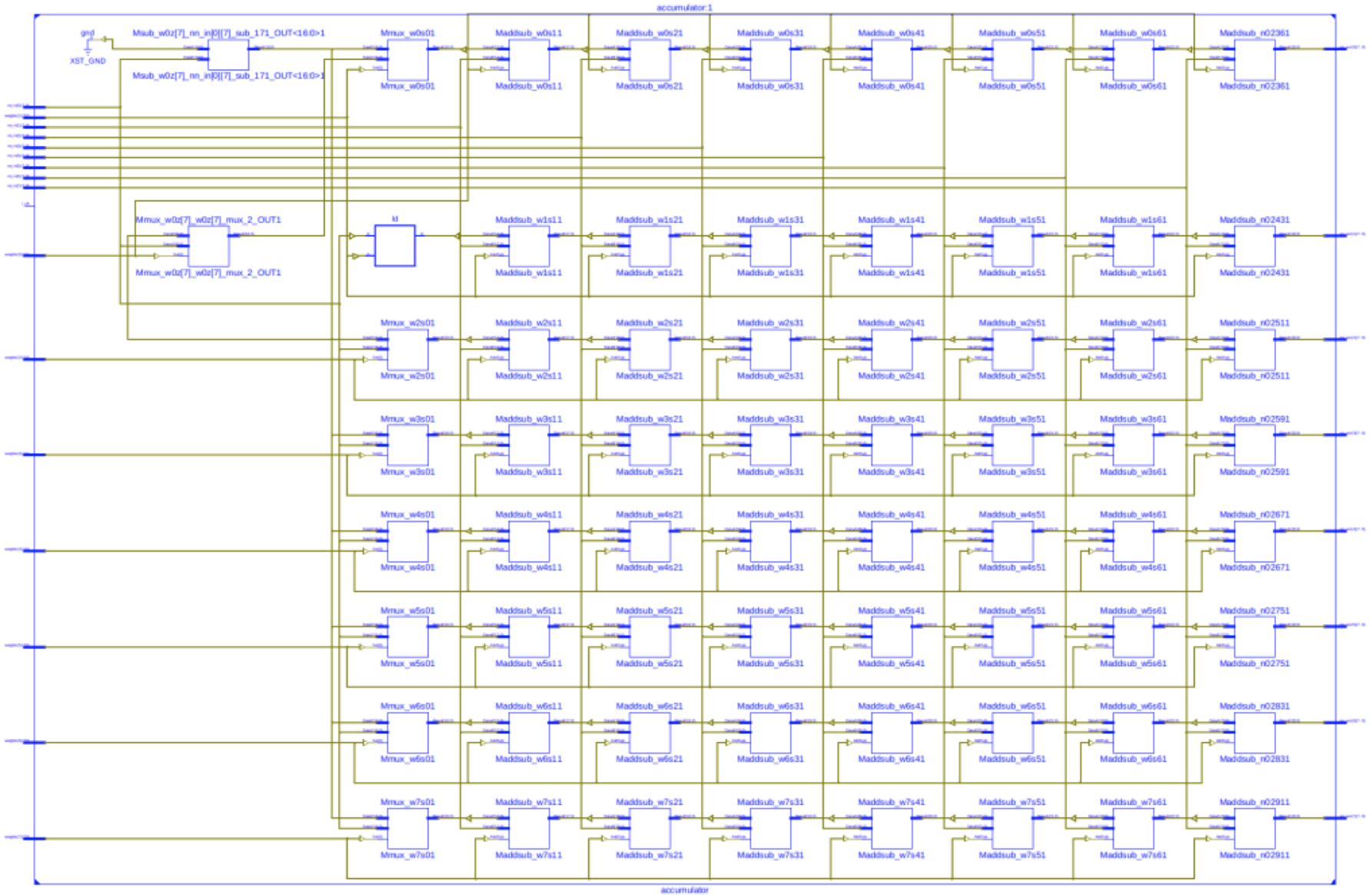
Computation mechanism

The following matrix multiplication(but for 8x1 vector) is computed in the following manner:

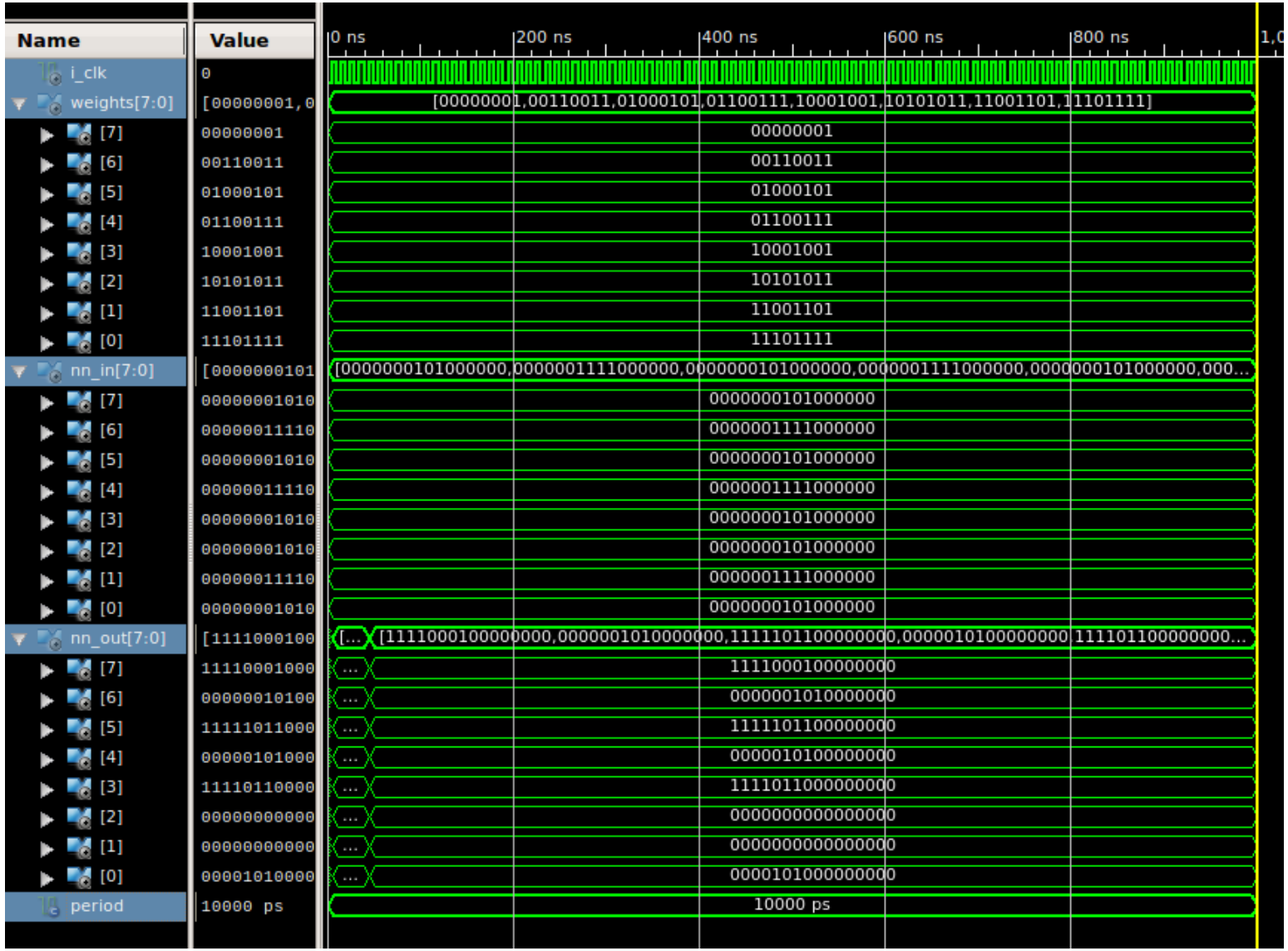
$$\begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 1.0 \\ 2.5 \\ 3.5 \end{bmatrix} = \begin{bmatrix} 1 - 2.5 + 3.5 \\ 1 + 2.5 - 3.5 \\ -1 - 2.5 - 3.5 \end{bmatrix}$$

```
if (weights(0)(0) = '1') then w0s0 <= w0z + nn_in(0); else w0s0 <= w0z - nn_in(0); end if;
if (weights(0)(1) = '1') then w0s1 <= w0s0 + nn_in(1); else w0s1 <= w0s0 - nn_in(1); end if;
if (weights(0)(2) = '1') then w0s2 <= w0s1 + nn_in(2); else w0s2 <= w0s1 - nn_in(2); end if;
if (weights(0)(3) = '1') then w0s3 <= w0s2 + nn_in(3); else w0s3 <= w0s2 - nn_in(3); end if;
if (weights(0)(4) = '1') then w0s4 <= w0s3 + nn_in(4); else w0s4 <= w0s3 - nn_in(4); end if;
if (weights(0)(5) = '1') then w0s5 <= w0s4 + nn_in(5); else w0s5 <= w0s4 - nn_in(5); end if;
if (weights(0)(6) = '1') then w0s6 <= w0s5 + nn_in(6); else w0s6 <= w0s5 - nn_in(6); end if;
if (weights(0)(7) = '1') then w0s7 <= w0s6 + nn_in(7); else w0s7 <= w0s6 - nn_in(7); end if;
nn_out(0)(7 downto -8) <= w0s7(7 downto -8);
```

Schematic


accumulator:1
gnd
XST_GND
Msub_w0z[7]_nn_in[0][7]_sub_171_OUT<16:0>1
Mmux_w0s01
Maddsub_w0s11
Maddsub_w0s21
Maddsub_w0s31
Maddsub_w0s41
Maddsub_w0s51
Maddsub_w0s61
Maddsub_n02361
Mmux_w0z[7]_w0z[7]_mux_2_OUT1
Id
Maddsub_w1s11
Maddsub_w1s21
Maddsub_w1s31
Maddsub_w1s41
Maddsub_w1s51
Maddsub_w1s61
Maddsub_n02431
Mmux_w2s01
Maddsub_w2s11
Maddsub_w2s21
Maddsub_w2s31
Maddsub_w2s41
Maddsub_w2s51
Maddsub_w2s61
Maddsub_n02511
Mmux_w3s01
Maddsub_w3s11
Maddsub_w3s21
Maddsub_w3s31
Maddsub_w3s41
Maddsub_w3s51
Maddsub_w3s61
Maddsub_n02591
Mmux_w4s01
Maddsub_w4s11
Maddsub_w4s21
Maddsub_w4s31
Maddsub_w4s41
Maddsub_w4s51
Maddsub_w4s61
Maddsub_n02671
Mmux_w5s01
Maddsub_w5s11
Maddsub_w5s21
Maddsub_w5s31
Maddsub_w5s41
Maddsub_w5s51
Maddsub_w5s61
Maddsub_n02751
Mmux_w6s01
Maddsub_w6s11
Maddsub_w6s21
Maddsub_w6s31
Maddsub_w6s41
Maddsub_w6s51
Maddsub_w6s61
Maddsub_n02831
Mmux_w7s01
Maddsub_w7s11
Maddsub_w7s21
Maddsub_w7s31
Maddsub_w7s41
Maddsub_w7s51
Maddsub_w7s61
Maddsub_n02911
accumulator

Waveform


Name
Value
0 ns
200 ns
400 ns
600 ns
800 ns
i_clk
0
weights[7:0]
[00000001,0
[00000001,00110011,01000101,01100111,10001001,10101011,11001101,11101111]
[7]
00000001
[6]
00110011
[5]
01000101
[4]
01100111
[3]
10001001
[2]
10101011
[1]
11001101
[0]
11101111
nn_in[7:0]
[0000000101
[0000000101000000,0000001111000000,0000000101000000,0000001111000000,0000000101000000,000...
[7]
00000001010
0000000101000000
[6]
00000011110
0000001111000000
[5]
00000001010
0000000101000000
[4]
00000011110
0000001111000000
[3]
00000001010
0000000101000000
[2]
00000001010
0000000101000000
[1]
00000011110
0000001111000000
[0]
00000001010
0000000101000000
nn_out[7:0]
[1111000100
[...
[1111000100000000,0000001010000000,1111011000000000,0000010100000000,1111011000000000...
[7]
11110001000
1111000100000000
[6]
00000010100
0000001010000000
[5]
11111011000
1111011000000000
[4]
00000101000
0000010100000000
[3]
11110110000
1111011000000000
[2]
00000000000
0000000000000000
[1]
00000000000
0000000000000000
[0]
00001010000
0000101000000000
period
10000 ps
10000 ps

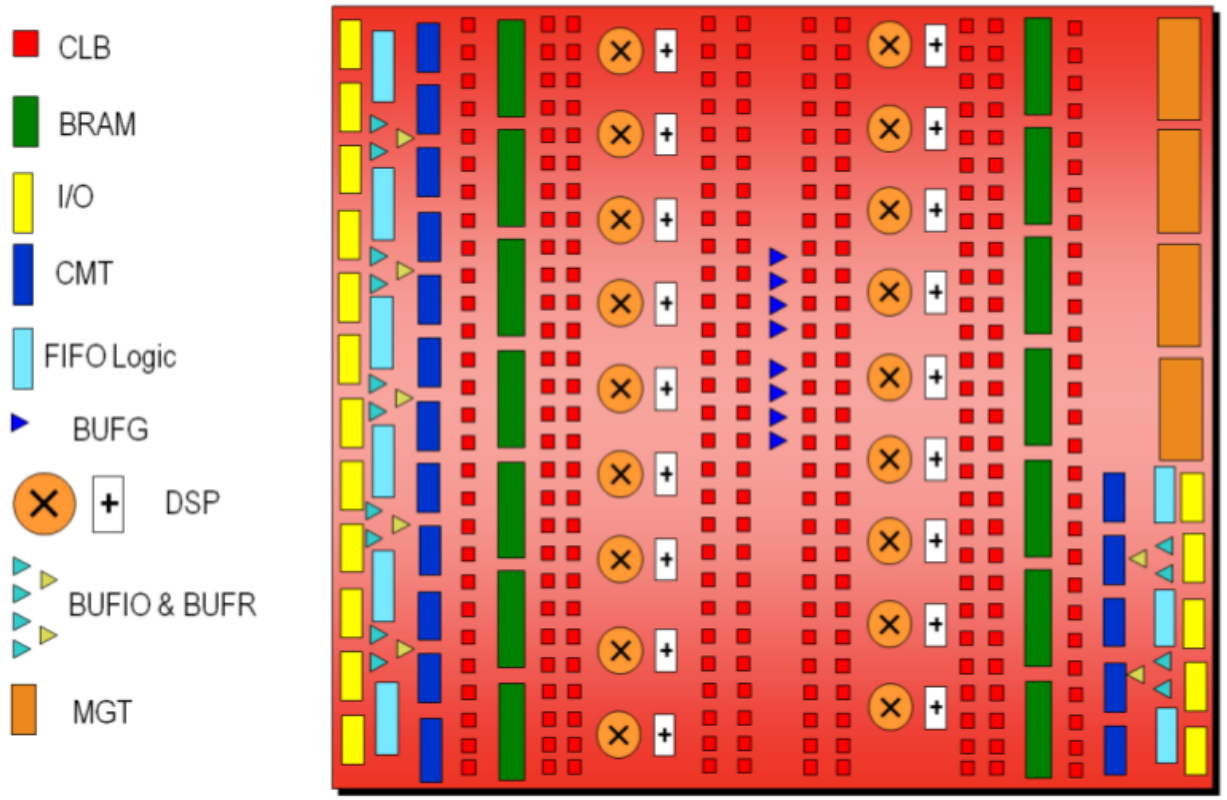
Scalar multiplier

- Implementation of a 32-bit fixed-point multiplication
  - 16-bit inputs: 1-bit signal, 7-bit integer, 8-bit fraction
- To perform fixed-point multiplication
  - Consider inputs as two-complements and do 'normal' multiplication, then determine where the "point" is.

1							1	1	0	0	1		-7
2	×						1	0	0	1	0		-14
3		0	0	0	0	0	0	0	0	0	0		
4		1	1	1	1	1	1	0	0	1			
5		0	0	0	0	0	0	0	0				
6		0	0	0	0	0	0	0					
7	+	0	0	0	1	1	1						
8		0	0	0	1	1	0	0	0	1	0		98

Source: https://www.allaboutcircuits.com/technical-articles/multiplication-examples-using-the-fixed-point-representation/

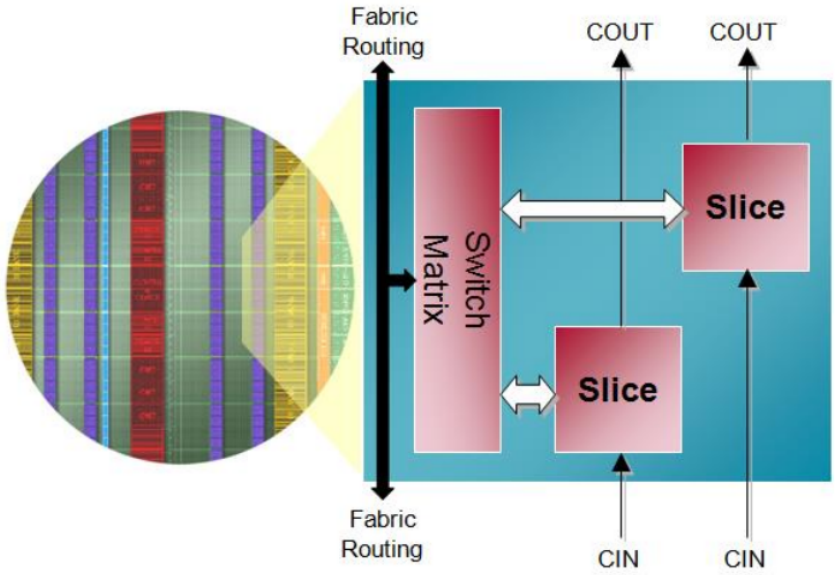
Xilinx 7 Series FPGA



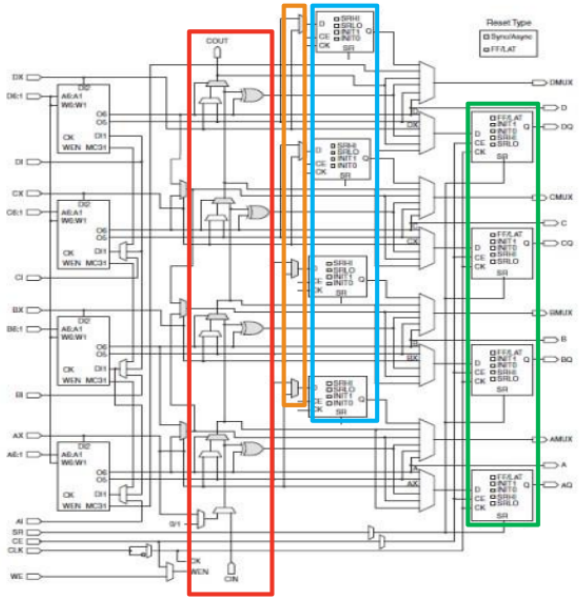
Artix-7 Architecture Overview

Xilinx 7 Series FPGA

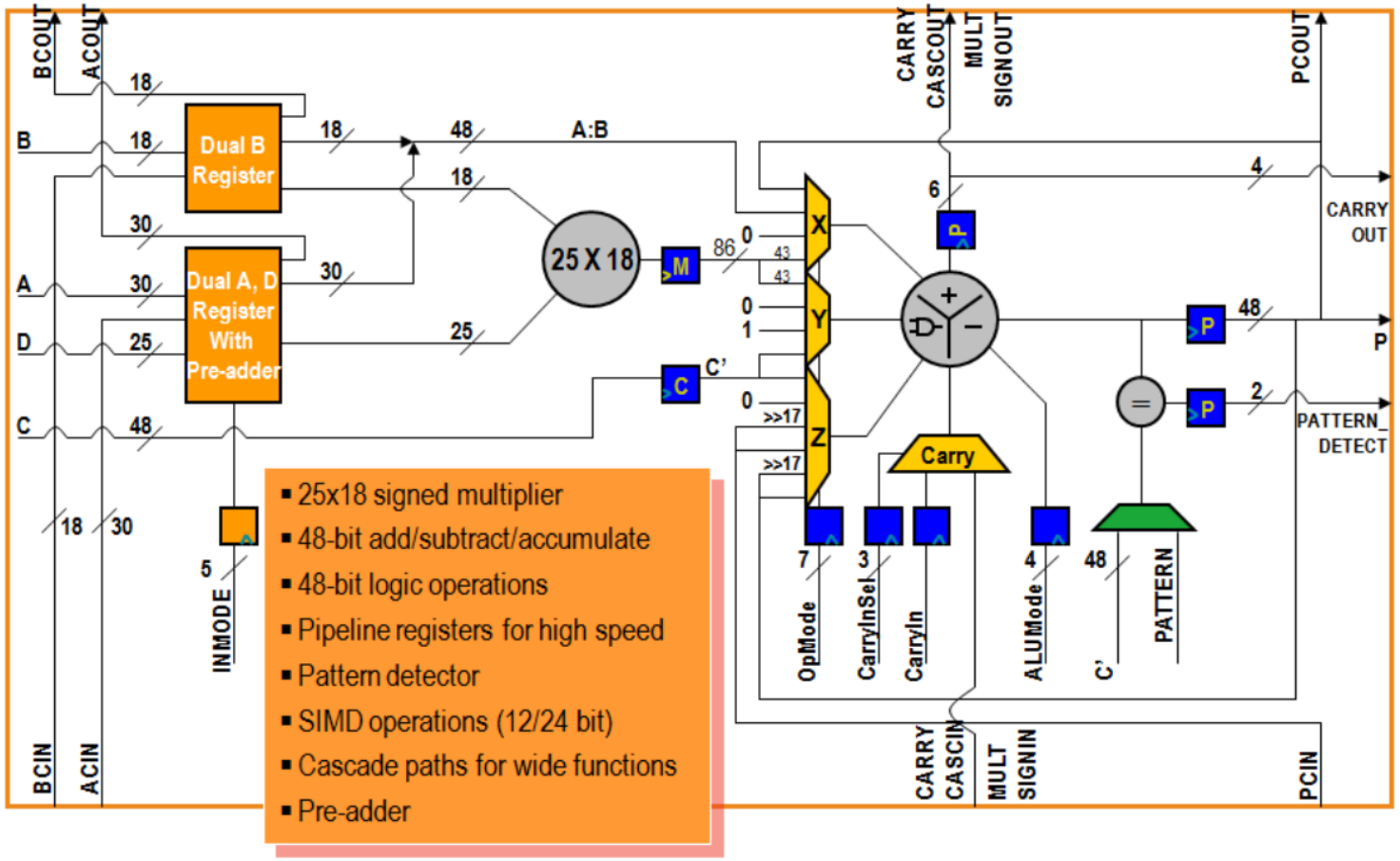
- Configurable Logic Block
  - Combinatorial functions
  - Flip-Flops



- Slices
  - Four six-input LUTs
  - Multiplexers
  - Carry Chains
  - Four Flip-Flops/Latches (plus another 4)



Xilinx 7 Series FPGA



Source: Xilinx

Remarks: 3-stage pipeline

Scalar Multiplier

- Besides VHDL version, we designed two more implementations to study resource allocations.
  - Verilog – small latency/area intensive
  - Verilog - DSP

Verilog: Area intensive vs DSP

- Area:

```
always @(scalar,in) begin
    if (~scalar[i])
        partial[i] <= 16'h0;
    else begin
        partial[i] <= tmp_in>>>(TMP_WIDTH-i);
    end
end

assign tmp_out =
      (partial[15]<<15)
    + (partial[14]<<14)
    + (partial[13]<<13)
    + (partial[12]<<12)
    + (partial[11]<<11)
    + (partial[10]<<10)
    + (partial[9]<<9)
    + (partial[8]<<8)
    + (partial[7]<<7)
    + (partial[6]<<6)
    + (partial[5]<<5)
    + (partial[4]<<4)
    + (partial[3]<<3)
    + (partial[2]<<2)
    + (partial[1]<<1)
    + partial[0];
```

Note: I indexes bits in 16-bit value

- DSP

```
always @(s_scalar, s_in_vector[i]) begin
    s_out_vector[i] <= s_in_vector[i] * s_scalar;
end
```

Note: I index 16-bit value in vector

Verilog: Area intensive vs DSP

- Area:

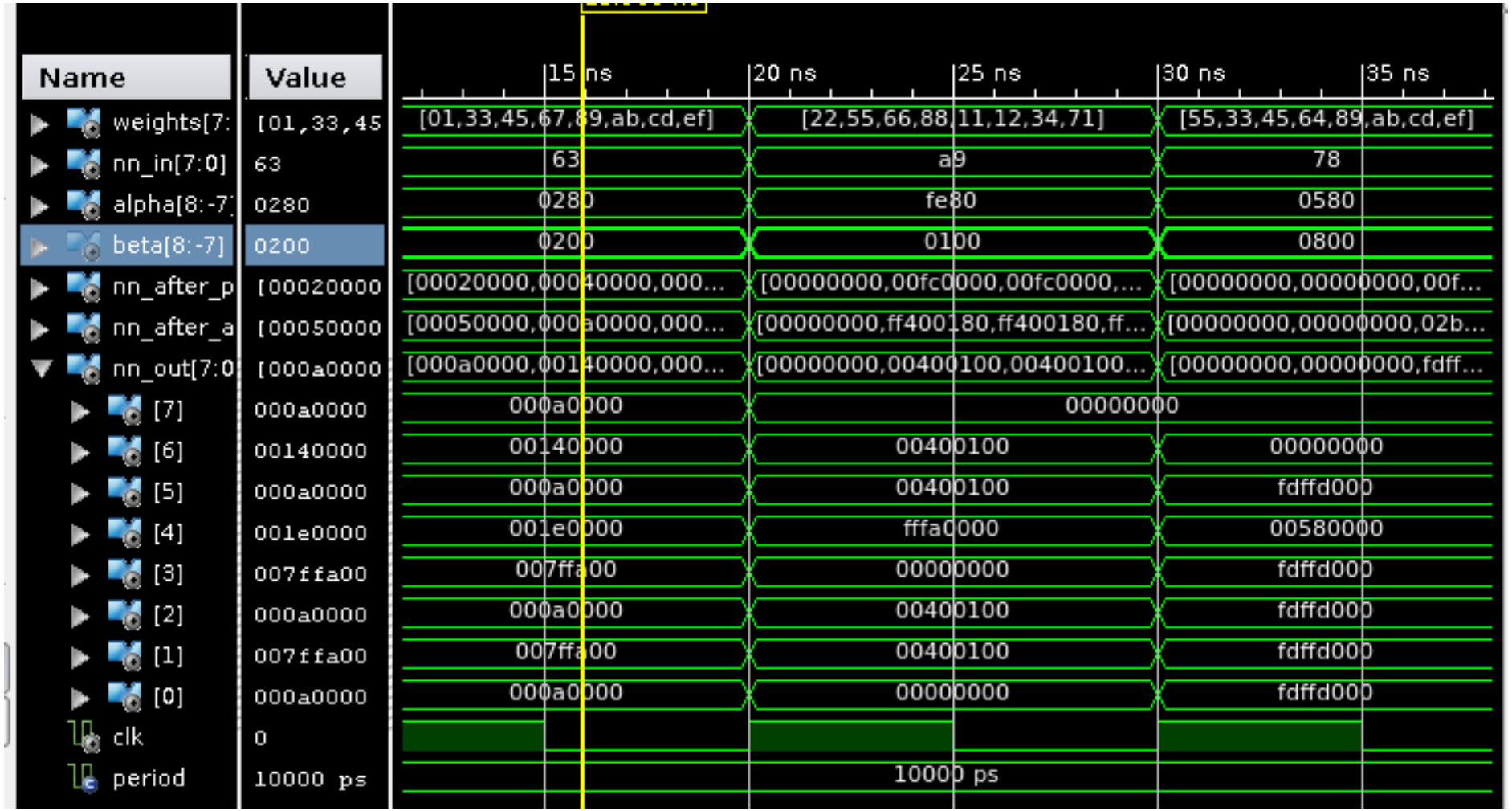
```
Slice Logic Utilization:
Number of Slice LUTs:                3288  out of  63400     5%
   Number used as Logic:             3288  out of  63400     5%
Slice Logic Distribution:
Number of LUT Flip Flop pairs used:   3288
  Number with an unused Flip Flop:    3288  out of   3288   100%
  Number with an unused LUT:             0  out of   3288     0%
  Number of fully used LUT-FF pairs:     0  out of   3288     0%
Multiplexers                                : 968
1-bit 2-to-1 multiplexer                    : 968
DSP
 Number of DSP48E1s:                   0  out of    240    0%
```

- DSP

```
Slice Logic Distribution:
  Number of LUT Flip Flop pairs used:      0
  Number with an unused Flip Flop:       0  out of      0
  Number with an unused LUT:             0  out of      0
Multiplexers =                              : 0
  1-bit 2-to-1 multiplexer                  : 0
DSP
   Number of DSP48E1s:                  24  out of    240    10%
```

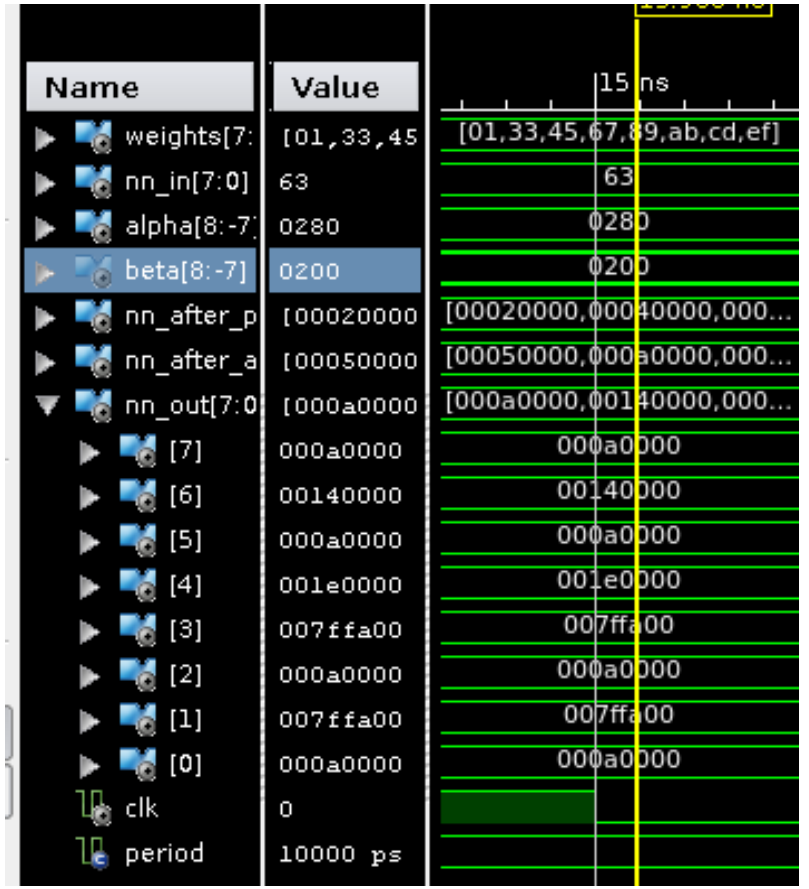
Results

XNOR-NET Results



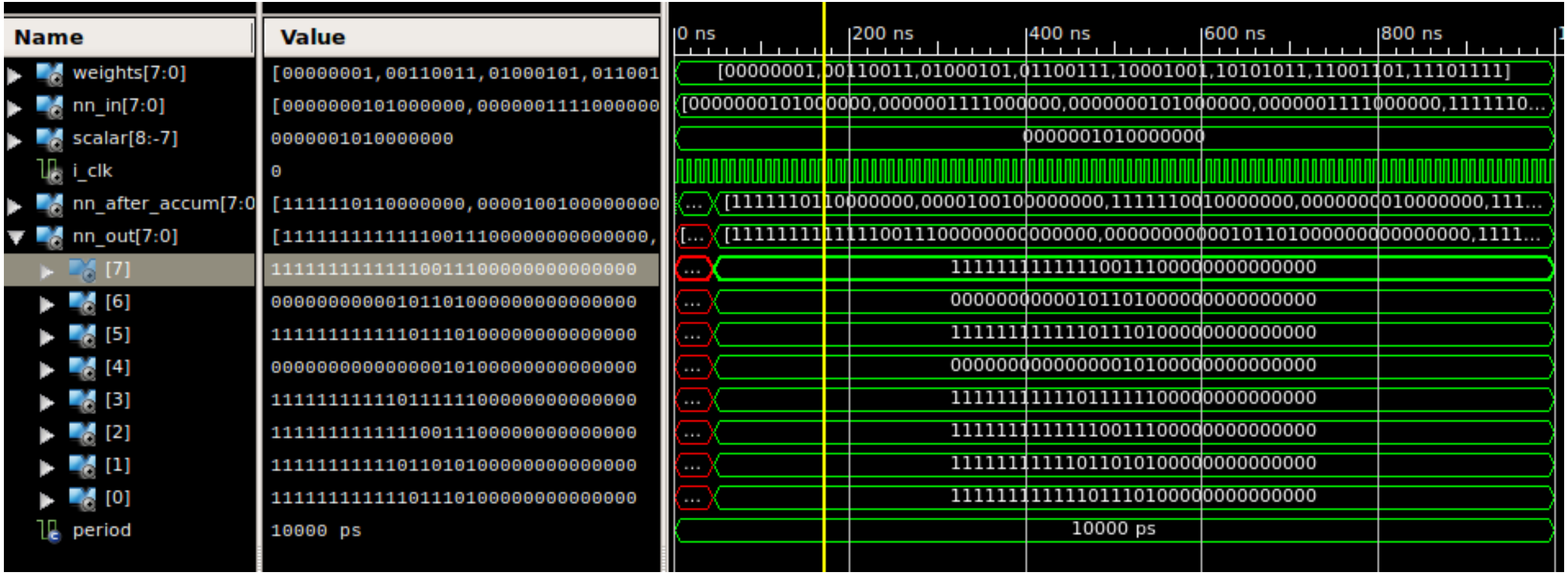
Python Verification of XNOR-Net

```python
import numpy as np

# Hyperparameters
batch_size = 8

# FC layer weight matrix
weights = np.asarray([0x01,0x33,0x45,0x67,0x89,0xAB,0xCD,0xEF])

# FC input after binarization
nn_in = 0x63
nn_out = 0x0

# Scalars after binarization
alpha = 2.5
beta = 2.0

# XNOR operation
xnor_results = np.zeros(batch_size)
for idx in range(batch_size):
    xnor_results[idx] = (nn_in ^ weights[idx])
print("xnor_results=",xnor_results)

# POPCOUNT operation
pop_results = np.zeros(batch_size)
for idx in range(batch_size):
    #print(str(bin(int(xnor_results[idx]))))
    num_ones = 8 - int(str(bin(int(xnor_results[idx]))).count("1"))
    pop_results[idx] = 2*num_ones-8
print("pop_results = ",pop_results)

# Scalar multiplication (alpha)
nn_results = np.zeros(batch_size)
for idx in range(batch_size):
    nn_results[idx] = pop_results[idx] * alpha
print("nn_results after alpha multiplication= ",nn_results)

# Scalar multiplication (beta)
for idx in range(batch_size):
    nn_results[idx] = nn_results[idx] * beta
print("nn_results after beta multiplication= ",nn_results)
```



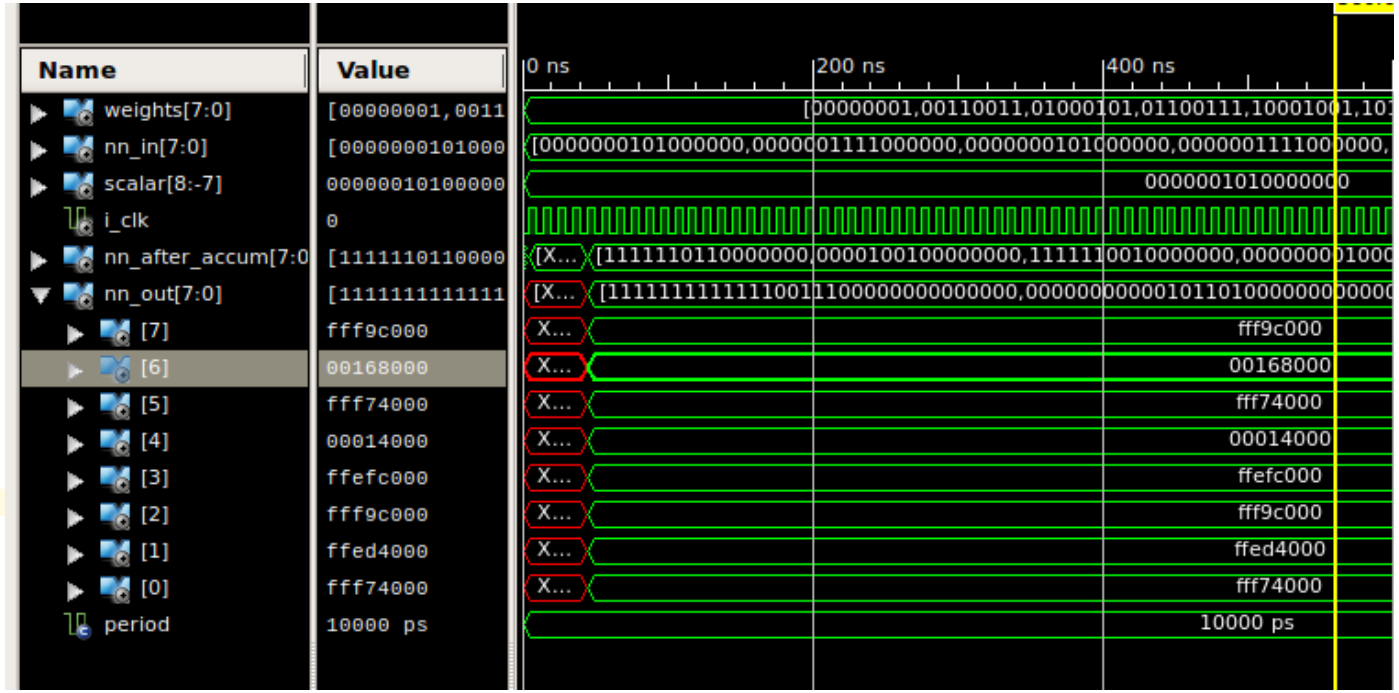

```
>>> reload(xnor_net)
('xnor_results=', array([  98.,   80.,   38.,    4.,  234.,  200.,  174.,  140.]))
('pop_results = ', array([ 2.,  4.,  2.,  6., -2.,  2., -2.,  2.]))
('nn_results after alpha multiplication= ', array([  5.,  10.,   5.,  15.,  -5.,   5.,  -5.,   5.]))
('nn_results after beta multiplication= ', array([ 10.,  20.,  10.,  30., -10.,  10., -10.,  10.]))
<module 'xnor_net' from 'xnor_net.pyc'>
```

Binary Connect Results



Python Verification of Binary Connect

```python
import numpy as np
from decimal import *

# Hyperparameters
batch_size = 8

# FC layer weight matrix
weights = np.asarray([0x01,0x33,0x45,0x67,0x89,0xAB,0xCD,0xEF])

# FC input after binarization
nn_in = [1.25, 3.75, 1.25, 3.75, -3.25, -4.25, 0.75, 0.75]
nn_out = list(np.zeros(8))

# Scalars after binarization
alpha = 2.5

# accumulation
weights = [format(x,'#010b')[2:] for x in weights]
for x in range(0,len(weights)):
    for y in range(0,len(weights[x])):
        if weights[x][y] == '1':
            nn_out[x] += nn_in[y]
        else:
            nn_out[x] -= nn_in[y]

print("Matrix vector multiplication after accumulator = ", nn_out)

nn_results = np.zeros(batch_size)
for idx in range(batch_size):
    nn_results[idx] = nn_out[idx] * alpha
print("nn_results after alpha multiplication = ",nn_results)
```




```
Matrix vector multiplication after accumulator =  [-2.5, 9.0, -3.5, 0.5, -6.5, -2.5, -7.5, -3.5]
nn_results after alpha multiplication =  [ -6.25  22.5   -8.75   1.25 -16.25  -6.25 -18.75  -8.75]
```