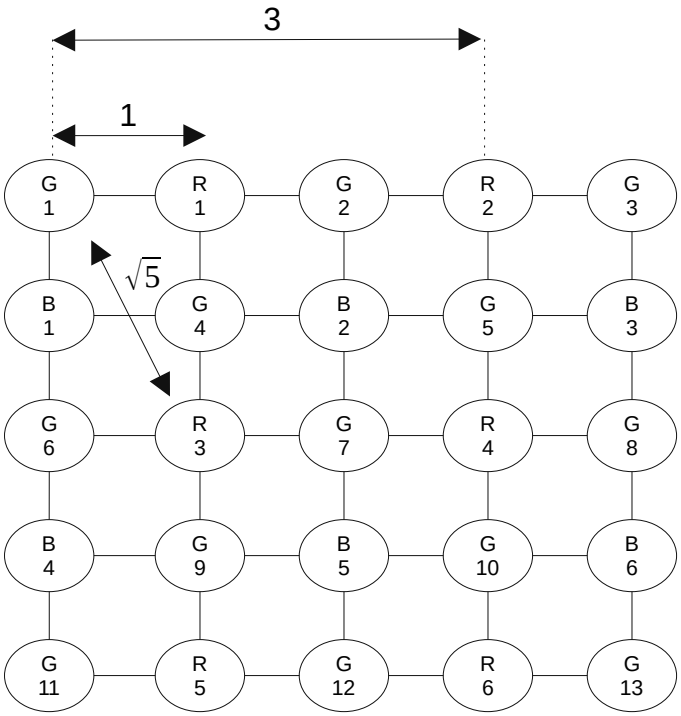


$$\frac{1}{1+\sqrt{5}+3}+\frac{\sqrt{5}}{1+\sqrt{5}+3}+\frac{3}{1+\sqrt{5}+3}=\frac{1+\sqrt{5}+3}{1+\sqrt{5}+3}=1$$

0.48107 0.35857 0.16035

$$\overline{R}=a_1\cdot\overline{R_1}+a_2\cdot\overline{R_2}+a_3\cdot\overline{R_3}$$

$$R^*=\overline{a_1}\cdot R_1+\overline{a_2}\cdot R_2+\overline{a_3}\cdot R_3$$

R * reads as R dual

$$R=0.48107\cdot\left(r_7^1\cdot 2^7+\ldots+r_0^1\cdot 2^0\right)+0.35857\cdot\left(r_7^3\cdot 2^7+\ldots+r_0^3\cdot 2^0\right)+0.16035\cdot\left(r_7^2\cdot 2^7+\ldots+r_0^2\cdot 2^0\right)$$

$$\overline{R}=0.48107\cdot\begin{pmatrix}r_7^1\\r_6^1\\r_5^1\\r_5^1\\r_4^1\\r_3^1\\r_2^1\\r_1^1\\r_0^1\end{pmatrix}+0.35857\cdot\begin{pmatrix}r_7^3\\r_6^3\\r_5^3\\r_5^3\\r_4^3\\r_3^3\\r_2^3\\r_1^3\\r_0^3\end{pmatrix}+0.16035\cdot\begin{pmatrix}r_7^2\\r_6^2\\r_5^2\\r_5^2\\r_4^2\\r_3^2\\r_2^2\\r_1^2\\r_0^2\end{pmatrix}$$

$$R^*=r_7^1r_6^1r_5^1r_4^1r_3^1r_2^1r_1^1r_0^1r_7^3r_6^3r_5^3r_4^3r_3^3r_2^3r_1^3r_0^3r_7^2r_6^2r_5^2r_4^2r_3^2r_2^2r_1^2r_0^2 \quad \text{call it} \quad R_{BIG}$$

Ex1 Px0, three surrounding red pixels

$$R_1=128_{10}=10000000_2$$
$$R_2=53_{10}=00110101_2$$
$$R_3=127_{10}=01111111_2$$
$$R_{BIG}=R_1R_3R_2=\underline{10000000}\,\underline{01111111}\,\underline{00110101},$$

24bit

$$R_{IDEAL}=0.48\cdot 128+0.36\cdot 127+0.16\cdot 53=115.64=01110011_2$$

$R=10000100_2=132$ not good approx.

■

Ex2 Px0, three surrounding red pixels

$R_1 = 128_{10} = 10000000_2$

$R_2 = 53_{10} = 00110101_2$

$R_3 = 127_{10} = 01111111_2$

$$R_{IDEAL} = 0.48 \cdot 128 + 0.36 \cdot 127 + 0.16 \cdot 53 = 115.64 = 01110011_2$$

Let $R_{12} = \overline{(R_1 >>_1 2) \oplus (R_2 <<_1 1)} = 01110100_2$,

where $>>_1$ and $<<_1$ - logical shift with '1' fill

$\oplus$ - XNOR boolean operator

- better approx. ■

Let $R = r_7 r_6 r_5 r_4 r_3 r_2 r_1 r_0$

Define Я as: $Я = r_4 r_5 r_6 r_7 r_0 r_1 r_2 r_3$ And define reversal operator Я as: $Я(R) = Я$

Ex3 From Ex2 $R_{12} = 01110100_2$

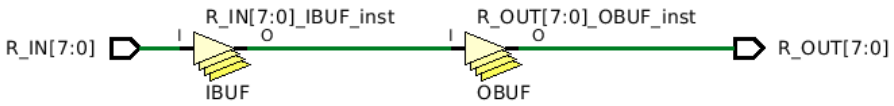
Let $Я_{12} = 11100010_2$

$\overline{Я_{12} \oplus (R_3 <<_1 1)} = 11100010_2$

$Я(\overline{Я_{12} \oplus (R_3 <<_1 1)}) = 01110100_2 = R_{12}$ - identity operator ■

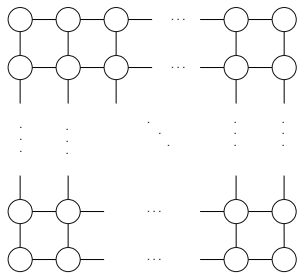
reverser_8b.vhd

```
entity reverser_8b is
  Port ( R_IN : in std_logic_vector(7 downto 0);
    R_OUT : out std_logic_vector(7 downto 0));
end reverser_8b;

architecture Dataflow of reverser_8b is

begin
   R_OUT(7) <= R_IN(4);
   R_OUT(6) <= R_IN(5);
   R_OUT(5) <= R_IN(6);
   R_OUT(4) <= R_IN(7);
   R_OUT(3) <= R_IN(0);
   R_OUT(2) <= R_IN(1);
   R_OUT(1) <= R_IN(2);
   R_OUT(0) <= R_IN(3);
end Dataflow;
```



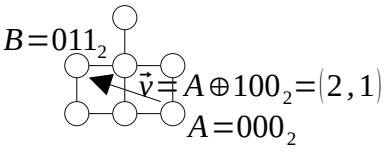
Recall $R^* = r_7^1 r_6^1 r_5^1 r_4^1 r_3^1 r_2^1 r_1^1 r_0^1 r_7^3 r_6^3 r_5^3 r_4^3 r_3^3 r_2^3 r_1^3 r_0^3 r_7^2 r_6^2 r_5^2 r_4^2 r_3^2 r_2^2 r_1^2 r_0^2$ call it R_{BIG}

Therefore R_{BIG} is a finite space and dual space R^* can be viewed as a rectangle or a plane consisting of 2^{24} elements in it arranged in a lattice pattern.



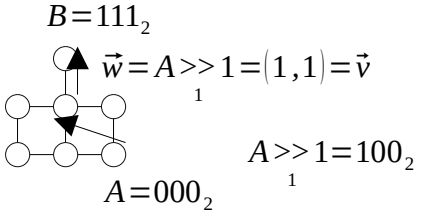
Thus we can define our binary operations such as AND,OR, etc as two different things:
1) Means of traversing the grid by flipping some of the bits
2) vectors
We choose one boolean operator to measure similarity and go from there. Let it be XNOR. Other operations will have to do with adjusting weights as we arrived here from the weighted sum and the end goal is weighted average.

Ex4

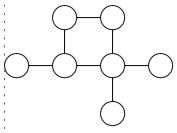


■

Ex5



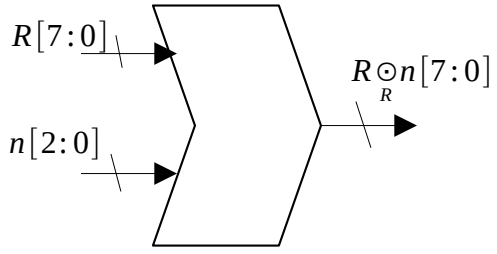
Therefore $\vec{v}, \vec{w}$ are orthogonal (hint from linear algebra) and these two operations (right shift with '1' fill and XNOR) span the entire lattice. ■



Can try different geometries/origin also. Actual notion of distance in this space may depend on the amount of such binary operations required to arrive from origin to any other point. Remember that we are looking for a point in this grid, that surely does exist, that is the closest approximation to the weighted average. Also remember once we contrived this space to be linear and dual by assuming 0.48 0.36 and 0.16 real numbers are independent (orthogonal) thus their products are equal to 0, all normal algebra went out the window.

Let $R = r_7 r_6 r_5 r_4 r_3 r_2 r_1 r_0$

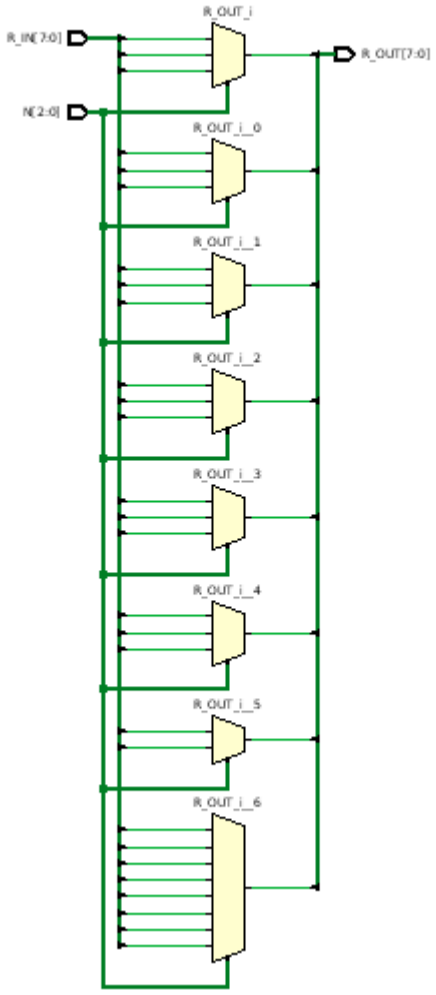
Define RPUSH operator as:



where n – control signal choosing which bit to push to the top position

rpush_8b.vhd

```
entity rpushn_8b is
  Port ( R_IN : in std_logic_vector(7 downto 0);
    N : in std_logic_vector(2 downto 0);
    R_OUT : out std_logic_vector(7 downto 0));
end rpushn_8b;

architecture Dataflow of rpushn_8b is

begin
     with N select
        R_OUT(7) <= R_IN(0) when "000",
          R_IN(1) when "001",
          R_IN(2) when "010",
          R_IN(3) when "011",
          R_IN(4) when "100",
          R_IN(5) when "101",
          R_IN(6) when "110",
          R_IN(7) when others;
      with N select
        R_OUT(6) <= R_IN(6) when "111",
          R_IN(7) when others;
      with N select
        R_OUT(5) <= R_IN(5) when "111",
          R_IN(5) when "110",
          R_IN(6) when others;
      with N select
        R_OUT(4) <= R_IN(4) when "111",
          R_IN(4) when "101",
          R_IN(5) when others;
      with N select
        R_OUT(3) <= R_IN(3) when "111",
          R_IN(3) when "100",
          R_IN(4) when others;
      with N select
        R_OUT(2) <= R_IN(2) when "111",
          R_IN(2) when "011",
          R_IN(3) when others;
      with N select
        R_OUT(1) <= R_IN(1) when "111",
          R_IN(1) when "010",
          R_IN(2) when others;
      with N select
        R_OUT(0) <= R_IN(0) when "111",
          R_IN(0) when "001",
          R_IN(1) when others;
end Dataflow;
```

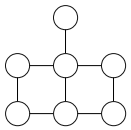


Ex6

Let $R = r_7 r_6 r_5 r_4 r_3 r_2 r_1 r_0$

$R \underset{R}{\odot} 3 = r_3 r_7 r_6 r_5 r_4 r_2 r_1 r_0$ ■

Ex7



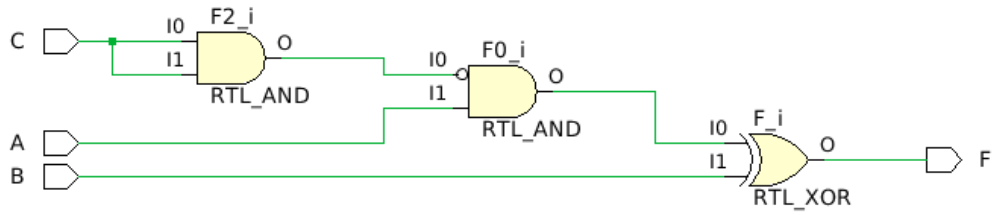
$\vec{u} = A \underset{R}{\odot} 0 = \vec{0}$

$A = 000_2$

RPUSH operator clearly doesn't do anything for the origin ■

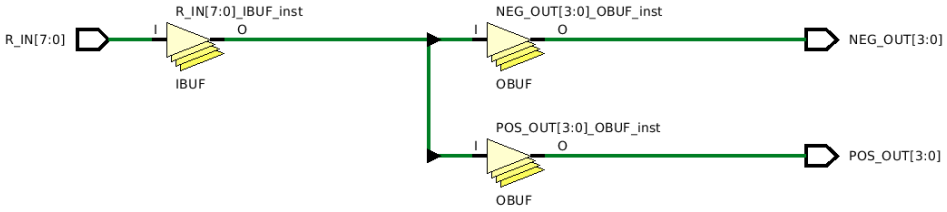
dim_1b.vhd

```
entity dim_1b is
  Port ( A : in std_logic;
    B : in std_logic;
    C : in std_logic;
    F: out std_logic);
end dim_1b;

architecture Dataflow of dim_1b is

begin
  F <= ((C NAND C) AND A) XOR B;

end Dataflow;
```



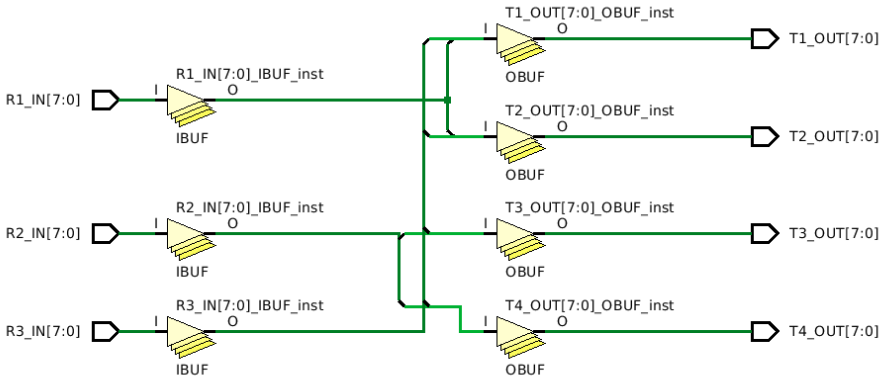
diode_8b.vhd

```
entity diode_8b is
  Port ( R_IN : in std_logic_vector(7 downto 0);
  POS_OUT : out std_logic_vector(3 downto 0);
  NEG_OUT : out std_logic_vector(3 downto 0));
end diode_8b;

architecture Dataflow of diode_8b is

begin
  POS_OUT(3) <= R_IN(7);
  POS_OUT(2) <= R_IN(6);
  POS_OUT(1) <= R_IN(1);
  POS_OUT(0) <= R_IN(0);
  NEG_OUT(3) <= R_IN(5);
  NEG_OUT(2) <= R_IN(4);
  NEG_OUT(1) <= R_IN(3);
  NEG_OUT(0) <= R_IN(2);

end Dataflow;
```

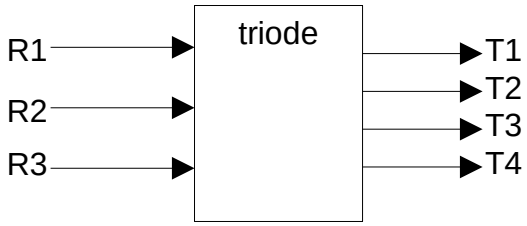


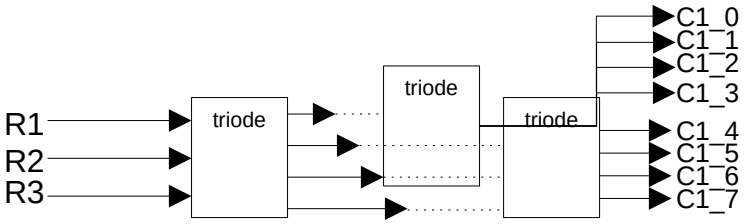
triode_8b.vhd

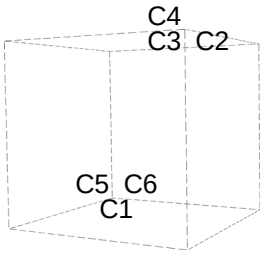
```
entity triode_8b is
  Port ( R1_IN : in std_logic_vector(7 downto 0);
  R2_IN : in std_logic_vector(7 downto 0);
  R3_IN : in std_logic_vector(7 downto 0);
  T1_OUT : out std_logic_vector(7 downto 0);
  T2_OUT : out std_logic_vector(7 downto 0);
  T3_OUT : out std_logic_vector(7 downto 0);
  T4_OUT : out std_logic_vector(7 downto 0));
end triode_8b;

architecture Dataflow of triode_8b is

begin
   T1_OUT(7) <= R1_IN(7);
   T1_OUT(6) <= R1_IN(6);
   T1_OUT(5) <= R3_IN(3);
   T1_OUT(4) <= R3_IN(4);
   T1_OUT(3) <= R3_IN(7);
   T1_OUT(2) <= R3_IN(0);
   T1_OUT(1) <= R1_IN(1);
   T1_OUT(0) <= R1_IN(0);

    T2_OUT(7) <= R1_IN(5);
    T2_OUT(6) <= R1_IN(4);
    T2_OUT(5) <= R3_IN(2);
    T2_OUT(4) <= R3_IN(5);
    T2_OUT(3) <= R3_IN(6);
    T2_OUT(2) <= R3_IN(1);
    T2_OUT(1) <= R1_IN(3);
    T2_OUT(0) <= R1_IN(2);

    T3_OUT(7) <= R2_IN(5);
    T3_OUT(6) <= R2_IN(4);
    T3_OUT(5) <= R3_IN(1);
    T3_OUT(4) <= R3_IN(6);
    T3_OUT(3) <= R3_IN(5);
    T3_OUT(2) <= R3_IN(2);
    T3_OUT(1) <= R2_IN(3);
    T3_OUT(0) <= R2_IN(2);

    T4_OUT(7) <= R2_IN(7);
    T4_OUT(6) <= R2_IN(6);
    T4_OUT(5) <= R3_IN(0);
    T4_OUT(4) <= R3_IN(7);
    T4_OUT(3) <= R3_IN(4);
    T4_OUT(2) <= R3_IN(3);
    T4_OUT(1) <= R2_IN(1);
    T4_OUT(0) <= R2_IN(0);

end Dataflow;
```



Series triodes. 8X6 matrices. Cubes.

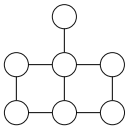






Six possible permutations of R1,R2,R3 fed into series triodes in the arrangement shown above give us a cube – unit of a three-dimensional lattice

Ex8



$A = 000_2$

1-bit triode: $Triode(R_1, R_2, R_3) = \begin{matrix} r_0^3 \\ r_0^1 \\ r_0^2 \\ r_0^3 \end{matrix}$ $Triode(R_1, R_3, R_2) = \begin{matrix} r_0^2 \\ r_0^1 \\ r_0^3 \\ r_0^2 \end{matrix}$

$R_1 = r_0^1 \quad R_2 = r_0^2 \quad R_3 = r_0^3$

$Triode(R_2, R_1, R_3) = \begin{matrix} r_0^3 \\ r_0^2 \\ r_0^1 \\ r_0^3 \end{matrix}$ $Triode(R_2, R_3, R_1) = \begin{matrix} r_0^1 \\ r_0^2 \\ r_0^3 \\ r_0^1 \end{matrix}$ $Triode(R_3, R_1, R_2) = \begin{matrix} r_0^2 \\ r_0^3 \\ r_0^1 \\ r_0^2 \end{matrix}$ $Triode(R_3, R_2, R_1) = \begin{matrix} r_0^1 \\ r_0^3 \\ r_0^2 \\ r_0^1 \end{matrix}$

=> 1-bit cube is $Cube(R_1, R_2, R_3) = \begin{matrix} r_0^2 & r_0^3 & r_0^1 & r_0^3 & r_0^1 & r_0^2 \\ r_0^3 & r_0^2 & r_0^3 & r_0^1 & r_0^2 & r_0^1 \\ r_0^1 & r_0^1 & r_0^2 & r_0^2 & r_0^3 & r_0^3 \\ r_0^2 & r_0^3 & r_0^1 & r_0^3 & r_0^1 & r_0^2 \\ r_0^3 & r_0^2 & r_0^3 & r_0^1 & r_0^2 & r_0^1 \\ r_0^1 & r_0^1 & r_0^2 & r_0^2 & r_0^3 & r_0^3 \\ r_0^2 & r_0^3 & r_0^1 & r_0^3 & r_0^1 & r_0^2 \\ r_0^3 & r_0^2 & r_0^3 & r_0^1 & r_0^2 & r_0^1 \end{matrix}$

■

Denoise. Piecewise linear functions.

Recall $\overline{R}=a_1\cdot\overline{R_1}+a_2\cdot\overline{R_2}+a_3\cdot\overline{R_3}$ is the value of the red component in the Bayer color filter array.

Introduce noise as a constant in our linear equation:

$\overline{R}=a_1\cdot\overline{R_1}+a_2\cdot\overline{R_2}+a_3\cdot\overline{R_3}+\overline{C}$,where C is noise

$$\overline{R}=0.48107\cdot\begin{pmatrix} r_7^1\\ r_6^1\\ r_5^1\\ r_5^1\\ r_4^1\\ r_3^1\\ r_2^1\\ r_1^1\\ r_0^1\end{pmatrix}+0.35857\cdot\begin{pmatrix} r_7^3\\ r_6^3\\ r_5^3\\ r_5^3\\ r_4^3\\ r_3^3\\ r_2^3\\ r_1^3\\ r_0^3\end{pmatrix}+0.16035\cdot\begin{pmatrix} r_7^2\\ r_6^2\\ r_5^2\\ r_5^2\\ r_4^2\\ r_3^2\\ r_2^2\\ r_1^2\\ r_0^2\end{pmatrix}+\begin{pmatrix} c_7\\ c_6\\ c_5\\ c_5\\ c_4\\ c_3\\ c_2\\ c_1\\ c_0\end{pmatrix}$$

Therefore now our Rbig looks like:

$R^*=r_7^1r_6^1r_5^1r_4^1r_3^1r_2^1r_1^1r_0^1r_7^3r_6^3r_5^3r_4^3r_3^3r_2^3r_1^3r_0^3r_7^2r_6^2r_5^2r_4^2r_3^2r_2^2r_1^2r_0^2c_7c_6c_5c_4c_3c_2c_1c_0$ call it R_{BIG}

Now, taking it back to the inherent orthogonality of flips and shifts and keeping in mind their vector nature. As this is our new basis, and it is clear that the order of operations is strict, i.e.

$$f\circ g\neq g\circ f$$

Naturally, interpret our red component vectors as:

$$i=\begin{Bmatrix}0\,induce\,shift\\ 1\,induce\,flip\end{Bmatrix}$$

Ex9

Px0, three surrounding red pixels

$R_1=128_{10}=10000000_2$ $R_1^*=F(S(S(S(S(S(S(x)))))))$

$R_2=53_{10}=00110101_2$ $R_2^*=S(S(F(F(S(F(S(F(x)))))))$

$R_3=127_{10}=01111111_2$ $R_3^*=S(F(F(F(F(F(F(x)))))))$

Notion of most and least significance of bits comes to help here.

Therefore our digital circuit core has to contain 2^8 = 256 dual vectors. ■

So what we really meant with R* and Rbig is:

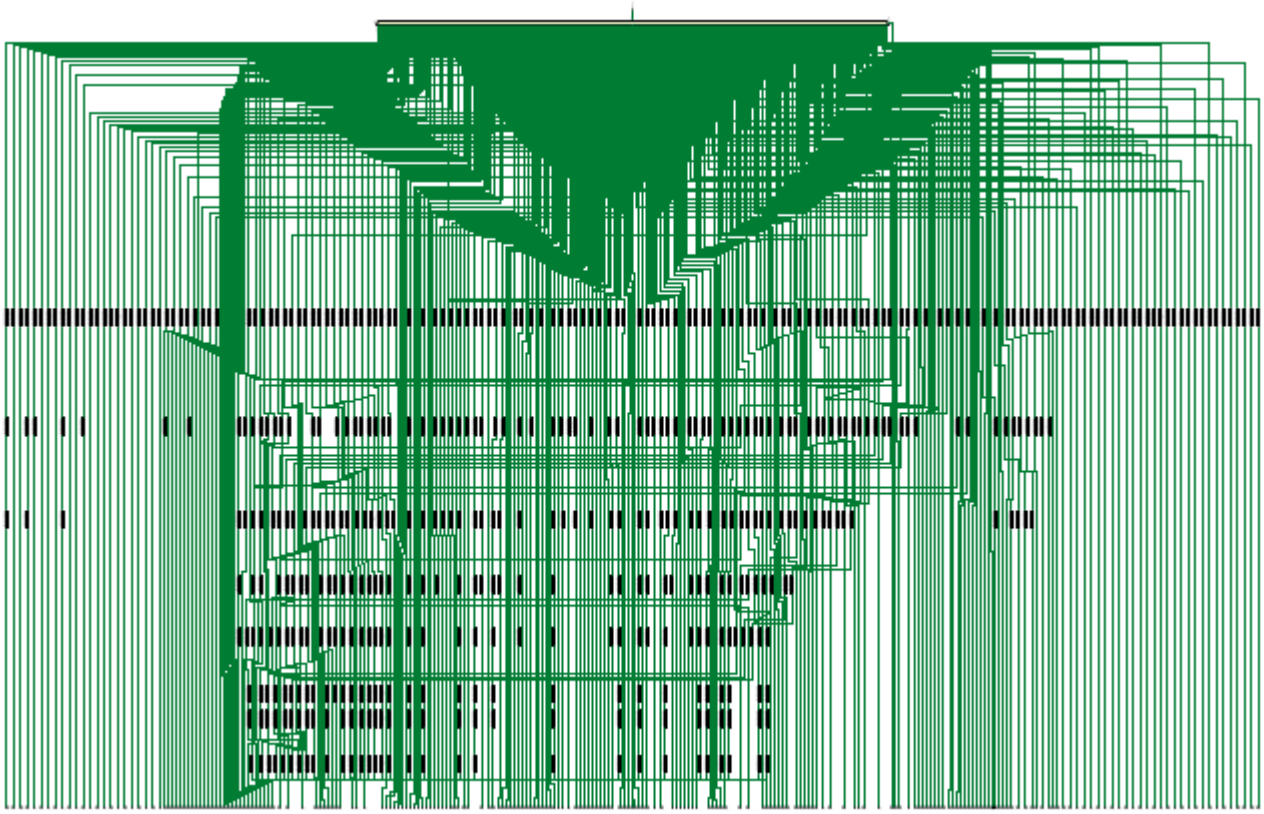
$R^*=r_7^1r_6^1r_5^1r_4^1r_3^1r_2^1r_1^1r_0^1\wedge r_7^3r_6^3r_5^3r_4^3r_3^3r_2^3r_1^3r_0^3\wedge r_7^2r_6^2r_5^2r_4^2r_3^2r_2^2r_1^2r_0^2$ call it R_{BIG} - wedge product of linear functionals.

dual_vector_core_8b.vhd

```
entity dual_vector_core_8b is
  Port ( X : in std_logic_vector(7 downto 0);
    D : in std_logic_vector(7 downto 0);
    OMEGA : out std_logic_vector(7 downto 0));
end dual_vector_core_8b;

architecture Dataflow of dual_vector_core_8b is

  --component declaration

  component flip_8b is
    port(D_IN : in std_logic_vector(7 downto 0);
      D_OUT : out std_logic_vector(7 downto 0));
  end component;

  component shift_8b is
    port(D_IN : in std_logic_vector(7 downto 0);
      D_OUT : out std_logic_vector(7 downto 0));
  end component;

  --component signals

  type w_t is array (0 to 255) of std_logic_vector(7 downto 0);
  signal w0 : w_t;
  signal w1 : w_t;
  signal w2 : w_t;
  signal w3 : w_t;
  signal w4 : w_t;
  signal w5 : w_t;
  signal w6 : w_t;
  signal w7 : w_t;
  signal w8 : w_t;

  signal decode_D : integer;

begin
   decode_D <= to_integer(unsigned(D));
   OMEGA <= w8(decode_D);

 GEN_DUAL:
   for I in 0 to 255 generate
     SHIFTX0 : if((I rem 2) = 0) generate
      S0 : shift_8b port map(
       D_IN => w7(I),
       D_OUT => w8(I)
     );
     end generate SHIFTX0;
     FLIPX0 : if((I rem 2) = 1) generate
      F0 : flip_8b port map(
       D_IN => w7(I),
       D_OUT => w8(I)
     );
     end generate FLIPX0;

     SHIFTX1 : if((I/2 rem 2) = 0) generate
      S1 : shift_8b port map(
       D_IN => w6(I),
       D_OUT => w7(I)
     );
     end generate SHIFTX1;
     FLIPX1 : if((I/2 rem 2) = 1) generate
      F1 : flip_8b port map(
       D_IN => w6(I),
       D_OUT => w7(I)
     );
     end generate FLIPX1;
     SHIFTX2 : if((I/4 rem 2) = 0) generate
      S2 : shift_8b port map(
       D_IN => w5(I),
       D_OUT => w6(I)
     );
     end generate SHIFTX2;
     FLIPX2 : if((I/4 rem 2) = 1) generate
      F2 : flip_8b port map(
       D_IN => w5(I),
       D_OUT => w6(I)
     );
     end generate FLIPX2;

     SHIFTX3 : if((I/8 rem 2) = 0) generate
      S3 : shift_8b port map(
       D_IN => w4(I),
       D_OUT => w5(I)
     );
     end generate SHIFTX3;
     FLIPX3 : if((I/8 rem 2) = 1) generate
      F3 : flip_8b port map(
       D_IN => w4(I),
       D_OUT => w5(I)
     );
     end generate FLIPX3;
```

dual_vector_core_8b.vhd (continued)

```
      SHIFTX4 : if((I/16 rem 2) = 0) generate
       S4 : shift_8b port map(
        D_IN => w3(I),
        D_OUT => w4(I)
      );
      end generate SHIFTX4;
      FLIPX4 : if((I/16 rem 2) = 1) generate
       F4 : flip_8b port map(
        D_IN => w3(I),
        D_OUT => w4(I)
      );
      end generate FLIPX4;

      SHIFTX5 : if((I/32 rem 2) = 0) generate
       S5 : shift_8b port map(
        D_IN => w2(I),
        D_OUT => w3(I)
      );
      end generate SHIFTX5;
      FLIPX5 : if((I/32 rem 2) = 1) generate
       F5 : flip_8b port map(
        D_IN => w2(I),
        D_OUT => w3(I)
      );
      end generate FLIPX5;
      SHIFTX6 : if((I/64 rem 2) = 0) generate
       S6 : shift_8b port map(
        D_IN => w1(I),
        D_OUT => w2(I)
      );
      end generate SHIFTX6;
      FLIPX6 : if((I/64 rem 2) = 1) generate
       F6 : flip_8b port map(
        D_IN => w1(I),
        D_OUT => w2(I)
      );
      end generate FLIPX6;

      SHIFTX7 : if((I/128 rem 2) = 0) generate
       S7 : shift_8b port map(
        D_IN => w0(I),
        D_OUT => w1(I)
      );
      end generate SHIFTX7;
      FLIPX7 : if((I/128 rem 2) = 1) generate
       F7 : flip_8b port map(
        D_IN => w0(I),
        D_OUT => w1(I)
      );
      end generate FLIPX7;

      w0(I) <= X;
    end generate GEN_DUAL;

end Dataflow;
```



1 dual vector = approx. 2000 logic cells

Ex10

Px0, three surrounding red pixels

$R_1 = 128_{10} = 10000000_2$
$R_2 = 53_{10} = 00110101_2$
$R_3 = 127_{10} = 01111111_2$

$F(x) = x \otimes 01000000_2$

$S(x) = x >>_0 1$

Compute $dx(0), dy(0), dz(0)$ in R* in a 8-bit binary.

Solution:

$dx(0) = F(S(S(S(S(S(S(0)))))))) = 01000000_2$
$dy(0) = S(S(F(F(S(F(S(F(0)))))))) = 00001100_2$
$dz(0) = S(F(F(F(F(F(F(0)))))))) = 00100000_2$

- Input method.

■

Ex11

Express $4\,dxdydz$ in the example above.

Solution:

$4\,dxdydz = dxdydz + dxdydz + dxdydz + dxdydz$

We need to know more about this space to be able to define what a constant ‘4’ is.

For ease of further computations, introduce BCD (binary coded digital) notation for shifts:

$S(x)=x>>_0 n=X\neg YYYY\ldots1\ldots YYY$ with Y = 0 and 1 in the nth place

Ex12

$S(x)=x>>_0 5=x\neg 00010000_2$ ■

Double dual V**

Recall from the original vectors $(0.48),(0.36),(0.16)$

$0.48\cdot256=01111010_2=122.88\approx123_{10}$
$0.36\cdot256=01011100_2=92.16\approx92_{10}$
$0.16\cdot256=00101000_2=40.96\approx41_{10}$

Vectors $\phi_1=(01111010)_2,\phi_2=(01011100)_2,\phi_3=(00101000)_2$ span the double dual space V**.

It was clear in Ex.10 that it is convenient for us to do calculations on this space at 0.

Annihilators

For some subset W of a vector space V, $\phi\in V^*$ is called an annihilator of W if

$$\phi(w)=0 \text{ for all } w\in W$$

The set of all such mappings W^0 is called the annihilator of W and is a subspace of V*.

Attributes:

i) $dim\,W+dim\,W^0=dim\,V$

ii) $W^{00}=W$, $where\,W^{00}=\{v\in V: v(\phi)=0\,for\,all\,\phi\in W^0\}$

$$v^i(\phi^j)=\begin{cases}1, i=j\\0, i\neq j\end{cases}$$

Therefore our original vectors are the annihilators – something that was not so obvious in real number algebra. The consequences of this are:

Ex13 Px0, three surrounding red pixels

$v_1=0.48\cdot256=01111010_2=122.88\approx123_{10}$
$v_2=0.36\cdot256=01011100_2=92.16\approx92_{10}$
$v_3=0.16\cdot256=00101000_2=40.96\approx41_{10}$

$R_1=128_{10}=10000000_2$ $\quad\phi^1=(F,S,S,S,S,S,S,S)$
$R_2=53_{10}=00110101_2$ $\quad\phi^2=(S,S,F,F,S,F,S,F)$
$R_3=127_{10}=01111111_2$ $\quad\phi^3=(S,F,F,F,F,F,F,F)$

$$v^1=\begin{pmatrix}S\\F\\F\\F\\F\\S\\F\\S\end{pmatrix},\quad v^2=\begin{pmatrix}S\\F\\S\\F\\F\\F\\S\\S\end{pmatrix},\quad v^3=\begin{pmatrix}S\\S\\F\\S\\F\\S\\S\\S\end{pmatrix}$$

$i=\begin{cases}0\,induce\,shift\\1\,induce\,flip\end{cases}$

(continued on next page)

Ex13

$v_1 = 0.48 \cdot 256 = 01111010_2 = 122.88 \approx 123_{10}$
$v_2 = 0.36 \cdot 256 = 01011100_2 = 92.16 \approx 92_{10}$
$v_3 = 0.16 \cdot 256 = 00101000_2 = 40.96 \approx 41_{10}$
$\phi^1 = (F, S, S, S, S, S, S, S)$
$\phi^2 = (S, S, F, F, S, F, S, F)$
$\phi^3 = (S, F, F, F, F, F, F, F)$

$$v^1 = \begin{pmatrix} S \\ F \\ F \\ F \\ F \\ S \\ F \\ S \end{pmatrix}, \quad v^2 = \begin{pmatrix} S \\ F \\ S \\ F \\ F \\ F \\ S \\ S \end{pmatrix}, \quad v^3 = \begin{pmatrix} S \\ S \\ F \\ S \\ F \\ S \\ S \\ S \end{pmatrix}$$

$$i = \begin{cases} 0 \; induce \; shift \\ 1 \; induce \; flip \end{cases}$$

$$v^1 \phi^1 = \begin{pmatrix} S \circ F \\ F \circ S \\ F \circ S \\ F \circ S \\ F \circ S \\ S \circ S \\ F \circ S \\ S \circ S \end{pmatrix} = 1 = F \quad v^1 \phi^2 = \begin{pmatrix} S \circ S \\ F \circ S \\ F \circ F \\ F \circ F \\ F \circ S \\ S \circ F \\ F \circ S \\ S \circ F \end{pmatrix} = 0 = S \quad v^1 \phi^3 = \begin{pmatrix} S \circ S \\ F \circ F \\ F \circ F \\ F \circ F \\ F \circ F \\ S \circ F \\ F \circ F \\ S \circ F \end{pmatrix} = 0 = S$$

$$v^2 \phi^1 = \begin{pmatrix} S \circ F \\ F \circ S \\ S \circ S \\ F \circ S \\ F \circ S \\ F \circ S \\ S \circ S \\ S \circ S \end{pmatrix} = 0 = S \quad v^2 \phi^2 = \begin{pmatrix} S \circ S \\ F \circ S \\ S \circ F \\ F \circ F \\ F \circ S \\ F \circ F \\ S \circ S \\ S \circ F \end{pmatrix} = 1 = F \quad v^2 \phi^3 = \begin{pmatrix} S \circ S \\ F \circ F \\ S \circ F \\ F \circ F \\ F \circ F \\ F \circ F \\ S \circ F \\ S \circ F \end{pmatrix} = 0 = S$$

$$v^3 \phi^1 = \begin{pmatrix} S \circ F \\ S \circ S \\ F \circ S \\ S \circ S \\ F \circ S \\ S \circ S \\ S \circ S \\ S \circ S \end{pmatrix} = 0 = S \quad v^3 \phi^2 = \begin{pmatrix} S \circ S \\ S \circ S \\ F \circ F \\ S \circ F \\ F \circ S \\ S \circ F \\ S \circ S \\ S \circ F \end{pmatrix} = 0 = S \quad v^3 \phi^3 = \begin{pmatrix} S \circ S \\ S \circ F \\ F \circ F \\ S \circ F \\ F \circ F \\ S \circ F \\ S \circ F \\ S \circ F \end{pmatrix} = 1 = F$$

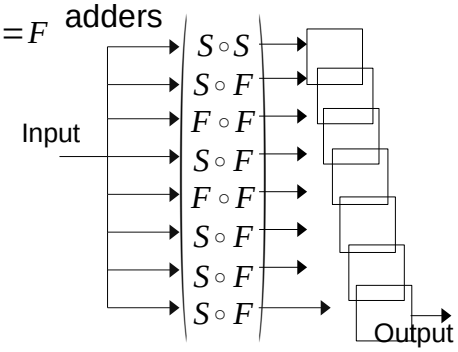
Superposition:
clearly as
$x_6(v^1 \phi^2) = F \circ S = 0$
$x_1(v^1 \phi^1) = F \circ S = 1$

Now consider
ex.11 again where we
had difficulty interpreting
constant 4 in 4dxdydz aka
$4\,dx \wedge dy \wedge dz$
Now it is clear that it is
just a vector
$4 = 00000100_2 = SSSSSFSS^2$

=> xdxdydz aka
$x dx \wedge dy \wedge dz$

is now a differentiable
and integrable function

Idea: for this to be a real
constant connect via
adders



So $v : V^* \to K$ and $\phi : V \to K$ for vector space V and its field K of scalars.

■

Just like all elements of $\mathbb{R}$ are unique, which allows us to do calculus on it, we construct this space such that it consists of unique elements.

Despite the unique results of the example above, they do not look as scalars.
We must remember the notion of differentiability of linear functions and rate of how the differential atomic elements such as dxdy tie together multiple dimensions.

For integrability on the lattice, we already found a function which is dual_vector_core.
Now we need to find the coefficient – the differential.

Thus:

$$i = \begin{Bmatrix} 0 \, if \; 0 \\ dxdy \, if \; 1 \end{Bmatrix}$$

Integration

Recall from Calculus an Integral is:

$$F(x)=\int_a^x f(t)\,dt \qquad \text{and 't' is a "dummy" variable} \tag{1}$$

Indefinite integral is:

$$\int f(x)\,dx=F(x)+C \qquad \text{,some constant C} \tag{2}$$

Recall that ω_i are functions and constants of linear functionals that we are working with:

$$\omega=\sum_{i_1<\ldots<i_k} \omega_{i_1,\ldots i_k} dx^{i_1}\wedge\ldots\wedge dx^{i_k} \tag{3}$$

Now we see that an Adder circuit is a special case and ADD C for some constant C satisfies the right side of equation (2):

$$F(x)+C=ADD\,C=\int f(x)\,dx \quad \text{for some function } f:\mathbb{R}\to\mathbb{R} \tag{4}$$

Differentiation

Recall from multivariate calculus that if $f:\mathbb{R}^n\to\mathbb{R}$ is differentiable, then

$$df=D_1 f\cdot dx^1+\ldots+D_n f\cdot dx^n$$

Partitions of unity

Although the notion may be same but in different domains, both integration and our initial problem of weighted averages has to do with partitions of unity. Traditionally, dx, dy, dz, and the like are considered to represent infinitesimal change in a variable which describes motion along one of the axis.

From our initial assumption, a unity interval $[0,1]\in\mathbb{R}$ is partitioned into three segments which we chose to be our basis vectors. Even then, with binary numbers partitioning the unity is hard – in each state of a bit we only may end up at either boundary of the unit interval. Therefore line integrals and Stoke's Theorem may suit us best in this situation:

$$\int_M d\,\omega=\int_{\partial M}\omega$$

Theorem says instead of integrating on the entire interval we may integrate on the boundary. Which is, 0 and 1 for a unity interval again in our favor. Since we already learned a lot about double dual space V** and naturally it's vectors take origin from our weighted partitioning of unity, integration will have to do with double dual vectors.

One wild assumption

Taking it all the way back to our original problem at hand, finding the weighted average:

$$\frac{1}{1+\sqrt{5}+3}+\frac{\sqrt{5}}{1+\sqrt{5}+3}+\frac{3}{1+\sqrt{5}+3}=\frac{1+\sqrt{5}+3}{1+\sqrt{5}+3}=1=F$$

This is another thing to be discovered.

Ex14

Px0, three surrounding red pixels

$R_1=128_{10}=10000000_2$ $\quad R_1^*=F(S(S(S(S(S(S(S(x)))))))$

$R_2=53_{10}=00110101_2$ $\quad R_2^*=S(S(F(F(S(F(S(F(x)))))))$

$R_3=127_{10}=01111111_2$ $\quad R_3^*=S(F(F(F(F(F(F(F(x)))))))$

Define

$F(x)=x\oplus 01000000_2=x\pm 64_{10}$

$S(x)=x>>_0 1=\frac{x}{2}$

Let's take a look at what this means from real number standpoint:

$$f_1(x)=\frac{x}{32}\pm 64$$

$$f_2(x)=\frac{\pm 64\pm 64+\frac{\pm 64+\frac{x\pm 64}{2}}{2}}{4}$$

$$f_3(x)=\frac{x\pm 448}{2}$$

At 0:

$$f_1(0)=\pm 64$$

$$f_2(0)=\frac{\pm 64\pm 64+\frac{\pm 64+\frac{\pm 64}{2}}{2}}{4}$$

$$f_3(0)=\frac{\pm 448}{2}$$

■

Eigenvectors and eigenvalues

Define $\lambda_i = R_i^*(R_i)$

Ex14 Px0, three surrounding red pixels

$R_1 = 128_{10} = 10000000_2$ $\quad R_1^* = F(S(S(S(S(S(S(S(x)))))))))$

$R_2 = 53_{10} = 00110101_2$ $\quad R_2^* = S(S(F(F(S(F(S(F(x)))))))))$

$R_3 = 127_{10} = 01111111_2$ $\quad R_3^* = S(F(F(F(F(F(F(F(x)))))))))$

$F(x) = x \otimes 01000000_2$

$S(x) = x >>_0 1$

$\lambda_1 = 01000001_2$

$\lambda_2 = 00001111_2$

$\lambda_3 = 00001111_2$

Taking original vectors:

$v_1 = 0.48 \cdot 256 = 01111010_2 = 122.88 \approx 123_{10}$

$v_2 = 0.36 \cdot 256 = 01011100_2 = 92.16 \approx 92_{10}$

$v_3 = 0.16 \cdot 256 = 00101000_2 = 40.96 \approx 41_{10}$

$\lambda_1(v_1) = 00100001_2 = 33_{10}$

$\lambda_2(v_2) = 00000101_2 = 5_{10}$

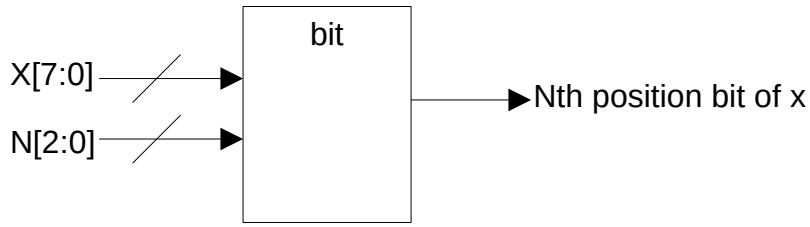
$\lambda_3(v_3) = 00001101_2 = 13_{10}$

■

Bit function

Define a bit function as:

$$Bit_n(x) = x_n$$



Define derivative as:

$$\frac{\partial M}{\partial F} = Bit^n \; if \; M \mid n = F$$

Matrices

Consider us altering our initial design of dual_vector_core_8b the following way:

```
--component declaration

component flip_8b is
  port(D_IN : in std_logic_vector(7 downto 0);
    D_OUT : out std_logic_vector(7 downto 0));
end component;

component shift_8b is
  port(D_IN : in std_logic_vector(7 downto 0);
    D_OUT : out std_logic_vector(7 downto 0));
end component;
```

=>

```
component flip_8b is
  port(D_IN : in std_logic_vector(7 downto 0);
    ENAB : in std_logic;
    D_OUT : out std_logic_vector(7 downto 0));
end component;

component shift_8b is
  port(D_IN : in std_logic_vector(7 downto 0);
    ENAB : in std_logic;
    D_OUT : out std_logic_vector(7 downto 0));
end component;
```

By supplying a 256x8 matrix M on the inputs that contain ENAB signals to each individual flip and shift element, differentiation as shown above is achieved.

Another benefit is that with additional wires this replaces what potentially could have been an n! (reads n factorial) size component to account for every possible ENABLE/DISABLE on each F or S circuit.

For an 8 bit system, 8! = 40320. In comparison, dual_vector_core is only 256 *8 = 2048 linear elements. We must account for size of our digital system, as at the moment of writing this, the highest capacity FPGA chip available on wholesale market has 8 million logic cells and costs like a wing of an airplane (really expensive in 2024).

Once again: differentiating with respect to a function = disabling that function.

Ex15

As we've previously figured:

$$f(x) = 4 = 00000100_2 = S(S(S(S(S(F(S(S(x))))))))$$

$$\frac{\partial f}{\partial F} = S(S(S(S(S(S(x)))))), \qquad \frac{\partial f}{\partial S} = F(x)$$

■

If we precompute some mathematic functions at points of interest and save the results as 8bit constants in a ROM that can give us instantaneous computation results.

Limits

As we take a lot of similarities from calculus, we need to remember that it was the properties of real numbers $\mathbb{R}:(-\infty,\infty)$ and intervals on real numbers such as $[0,1]$ that allowed all the well-known results to be achieved. But now it is our turn to be presented with a question: what is the number space that we are working in that we want to obtain useful results in, aside from symbol-pushing? Our results so far may look interesting and nice but they need to be applied to something that makes sense to whoever applies.

Recall that we were presented with a problem of interpolating the value of the red pixel from it's three unequally distant neighbors (closest, closer, and farthest one) within the square grid covered with a Bayer color filter array, such as the one that comes with all cameras and image sensors. Each cell in the grid therefore can only hold value for one of the colors – red, green or blue. To be able to send the picture to a monitor we need to know the intensity of other two colors of each pixel.

So we just accidentally answered our own question. It is the intensity or the power that our 8-bit value that we read from the image sensor is.

0 = no power = no component of this color in this pixel
255 = full power = component of this color is very strong in this pixel
and everything in between, producing the interval $[0,255]\in\mathbb{N}$

However, it is not the only possible way to interpret 8-bit binary number. Another well-known way is to treat the MSB (most significant bit) as the sign bit, therefore centering 256 possible values around 0 and half of them, 128 will be with the negative sign, producing the interval $[-128,127]\in\mathbb{N}$

Yet another useful way, especially for the differential approach we are attempting to take would be to treat values above some 8bit number as approaching infinity.

Ex16

Let values from 128 and on to represent infinity (anything with MSB = '1' is infinity)

i.e $x\geqslant 10000000_2 \rightarrow \infty$

This way we obtain a vague but very real $[0,\infty)\in\mathbb{R}$ where the power of flipping each bit is up to us to decide. We may even normally work with values 0 to 127 and then say that starting at $x\geqslant 10000000_2 \rightarrow \infty$ it immediately shot up toward infinity.

double_dual_scalar_core_8b.vhd $(1, \sqrt{5}, 3)$ -- x8 required to reconstruct the vector