Efficient Data Extraction Circuit for Posit Number System: LDD-Based Posit Decoder

Team Members:

1. K.S.V. Rohit
   IMT2022576
   komaragiri.sai@iiitb.ac.in

2. A. Nishith
   IMT2022556
   a.nishith@iiitb.ac.in

3. A. Lokesh
   IMT2022577
   lokeshkumar.aravapalli@iiitb.ac.in

4. M. Lohitaksh
   IMT2022536
   lohitaksh.maruvada@iiitb.ac.in

5. T.V.S Chaitanya
   IMT2022545
   tvs.chaitanya@iiitb.ac.in

6. G. Pradyumna
   IMT2022555
   pradyumna.g@iiitb.ac.in

Introduction

Posit is a type III universal number (unum) that, along with two previously known parameters – number size (N) and exponent size (es), as shown in Fig. 1, is composed of four components in its representation:

1. Sign bit (s)
2. Regime bits (r)
3. Exponent bits (e)
4. Fraction bits (f),

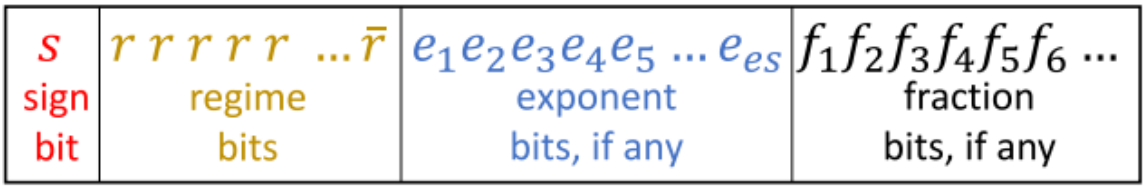


Fig. 1. Generic posit format for finite, nonzero values.

The length of the regime bits can vary which may even take over the space of fraction bits and exponent bits for different number values. This key property yields the tradeoff between decimal accuracy and dynamic range. However, it requires an extra decoding/data extraction process to obtain the sizes and values for each component before arithmetic calculation.

Posit vs IEEE 754:

1. **Range and Precision:**

Posit:

- Posits dynamically adjust their range and precision based on the value, which provides an efficient representation of both small and large numbers within the same bit width.
- Posits provide higher precision near zero and for values close to 1 due to their dynamic precision. They allocate bits more effectively, offering more

precision where it's most needed (**Tapered accuracy** (Explained in next section)).

IEEE 754:

- The range and precision are determined by fixed exponent and mantissa sizes, which can be less adaptable.
- Precision is fixed by the format's structure, with separate allocation for exponent and mantissa, which can lead to inefficient bit usage for certain values.

2. **Special Cases:**

Posit:

- Posits reduce exceptions by eliminating NaN (Not a Number). All bit patterns represent valid numbers, simplifying error handling in hardware and improving efficiency.
- When performing operations that traditionally result in NaN in floating-point arithmetic (such as dividing by zero or taking the square root of a negative number), posits handle these cases by returning ±∞. This avoids the logical contradiction of declaring something "unrepresentable" and then trying to represent it
- ±∞ is used as a default result, allowing the computation to continue while signaling that an error occurred.

IEEE 754:

- IEEE 754 reserves several bit patterns for special values like NaN, ±∞, and subnormal numbers, which increases complexity in hardware and computation.
- Invalid operations such as division by zero or the square root of a negative number produce NaN or ±∞, with NaN indicating that a result is undefined or unrepresentable.

3. **Hardware Complexity:**

Posit:

- Simpler hardware design in some cases because it eliminates special cases like NaN and infinities.
- Requires more complex decoding logic for regime and variable-length fields
- Not yet as widely adopted in hardware compared to IEEE 754.

IEEE 754:

- Widely implemented in hardware with mature support in CPUs, GPUs, and FPGAs.
- Circuit complexity arises due to normalization and handling edge cases (e.g., NaN, infinities, subnormals).

Tapered Accuracy Illustration:

- In the posit number system, the allocation of bits for different components (sign, regime, exponent, and fraction) dynamically adapts based on the magnitude of the number.
- This property of "tapered accuracy" ensures that numbers closer to 1 in magnitude are represented with higher precision, while extremely large or small numbers have lower precision.

Example:

Consider a 16-bit posit number (n=16) with es=2 (up to 2 exponent bits).

1. Number near 1:

- Example: 1.5
- Posit representation focuses most bits on precision near 1.
- The regime component is small (likely just one bit indicating the range).
- Exponent has up to 2 bits, giving precise scaling.
- Fraction has many bits for high precision (e.g., 12-13 bits).
- This allows accurate representation of small differences, like 1.5001.

2. Large number:

- Example: 256
- Posit representation focuses more bits on the regime to express the large scale.
- The regime may take up more bits to indicate the size range (e.g., 5-6 bits).
- Exponent may still use 2 bits but leave fewer bits for the fraction.
- Fraction has fewer bits, so precision is lower compared to numbers near 1.

3. Small number:

- Example: 0.00390625 (1/256)
- Similar to large numbers, small numbers require more regime bits to express the scale downwards.

- Less precision is allocated for the fraction.

Conclusion:

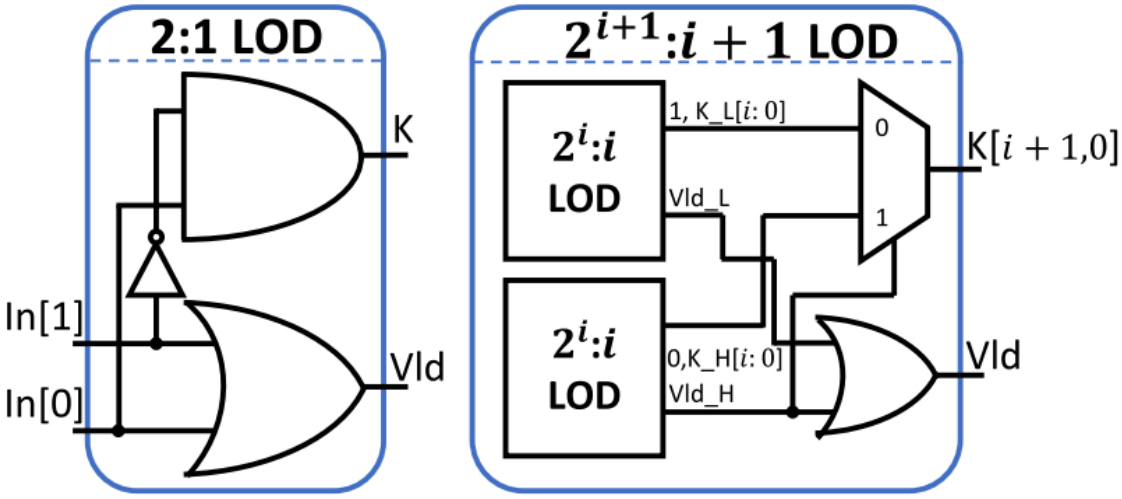
- **Numbers near 1**: Allocated bits = 1 sign + small regime + 2 exponent + many fraction bits → **High accuracy**.
- **Large numbers (256)**: Allocated bits = 1 sign + large regime + 2 exponent + few fraction bits → **Low accuracy**.
- **Small numbers (1/256)**: Allocated bits = 1 sign + large regime + 2 exponent + few fraction bits → **Low accuracy**.

Posit Data Extraction with Leading One's Detection Algorithm

- Let the predetermined values be N - size of the binary string (IN) and ES - the number of exponent bits.
- Now, the first bit has to be the sign bit.
- Let s = IN[N-1]. Now, if s is 0 it denotes a positive sign else a negative sign.
- If all the bits of IN are zero, then it denotes zero (z = NOR(IN[N-1:0]).
- If the sign bit of IN is 1 and the rest of the bits are zero, then it denotes infinity (±∞ as mentioned in "Special cases" section) (inf = IN[N-1] & NOR(IN[N-2:0]).
- If s is 0, then let XIN be IN[N-2:0] (where "XIN" is a temp variable used to store "IN" without sign bit).
- If s is 1, we need to take the 2's complement of the input string after the sign bit. So, XIN = ~IN[N-2:0] + 1.
- Now, we are using Leading One's Detection Algorithm. So, we need to check for ones in the binary string. If XIN[N-2] is 0, then LIN = XIN (where "LIN" is temp variable used to calculate position of leading one).
- If XIN[N-2] is 1, then we need to invert the bits. So, LIN = ~XIN.
- Then, we pass this LIN to the **Leading One Detector** Circuit which gives us the index of the first one in the input from MSB side, equal to K.
- If the first bit of XIN is 1, then the regime value –> r = K-1.
- If the first bit of XIN is 0, then the regime value is 2's complement of K – r = ~(K-1).
- Let's left shift XIN by K+1 bits and assign this value to temporary variable temp. So, temp = XIN<<(K+1). This removes all the regime bits and leaves the exponent and fractional bits.
- Now, if N – 2 – K is greater than ES, then the next ES bits are the expo bits, else whatever remaining bits are the expo bits with no fractional bits. So, e = the highest ES bits of temp from the MSB side.
- Now, if we remove the exponent bits by left shifting by ES bits, the remaining bits are the fractional bits. So, f = temp<<ES.
- Now, since we get the r, e and f values, we can calculate the decimal value of the number using the formula below.

$$\left(2^{2^{es}}\right)^{r} \times 2^{e} \times 1.f.$$

Leading One Detector:



2:1 LOD:

We need to find the position of occurrence of 1. If we write the K-map, we will get the same circuit as shown above for the 2:1 LOD. If '1' is not there among these 2 bits, the Vld will be LOW; if there is any, the valid will be high. The K will indicate the position of the 1 in those 2 bits,

N:1 LOD:

- To find the k value (the position of the first occurrence of '1') of an N-bit number, we first divide the bits into two sets of N/2(higher bits[N-1:N/2]/MSB) and N/2(lower bits[N/2-1:0]/LSB). The N should be a power of '2', if not, we will append ones at the end of the input number.
- Let's say we have divided them into two sets; we will get K and Vld from both of them, now we need to select the k from k_high and k_low using Vld_high as the select line.
- If Vld_h is high, the result of lower N/2 bits does not affect the value of k(the first occurrence of k), the higher bits are always before the lower bits. So, we take the same value of k_h of i bits and then append the ~(Vld_h) to the MSB side of the k to make it K+1 bits.
- If Vld_high is low, it means there is no one in half of the bits, so we select k_low and then append ~(Vld_h) to make it i+1 bits.
- This process is similar to converting a decimal number into its binary representation.

Example LOD Decoding process

. LOD

$g_{n=1}]0001011_2$

$s = 1$

⇒ We need to take 2's complement

⇒ $xin = \overline{0001011} + 1$

$= 1110100 + 1$

$=]1110101_2$

First bit of xin is 1

⇒ We need to take 1's complement of xin

⇒ $lin = \overline{1110101} = 0001010_2$

⇒ $K = 3$ (as 1 is at the 3rd index)

First bit of xin is 1

⇒ $r = K - 1 = 2 = 10_2$

$temp = 1110101 << 4 = 1010000_2$

Here since $N - K - 2 = 3 > ES = 1$

⇒ e = Highest ES bits of temp

⇒ $e = temp[6] = 1_2$

$f = 1010000_2 << 1 = 0100000_2$

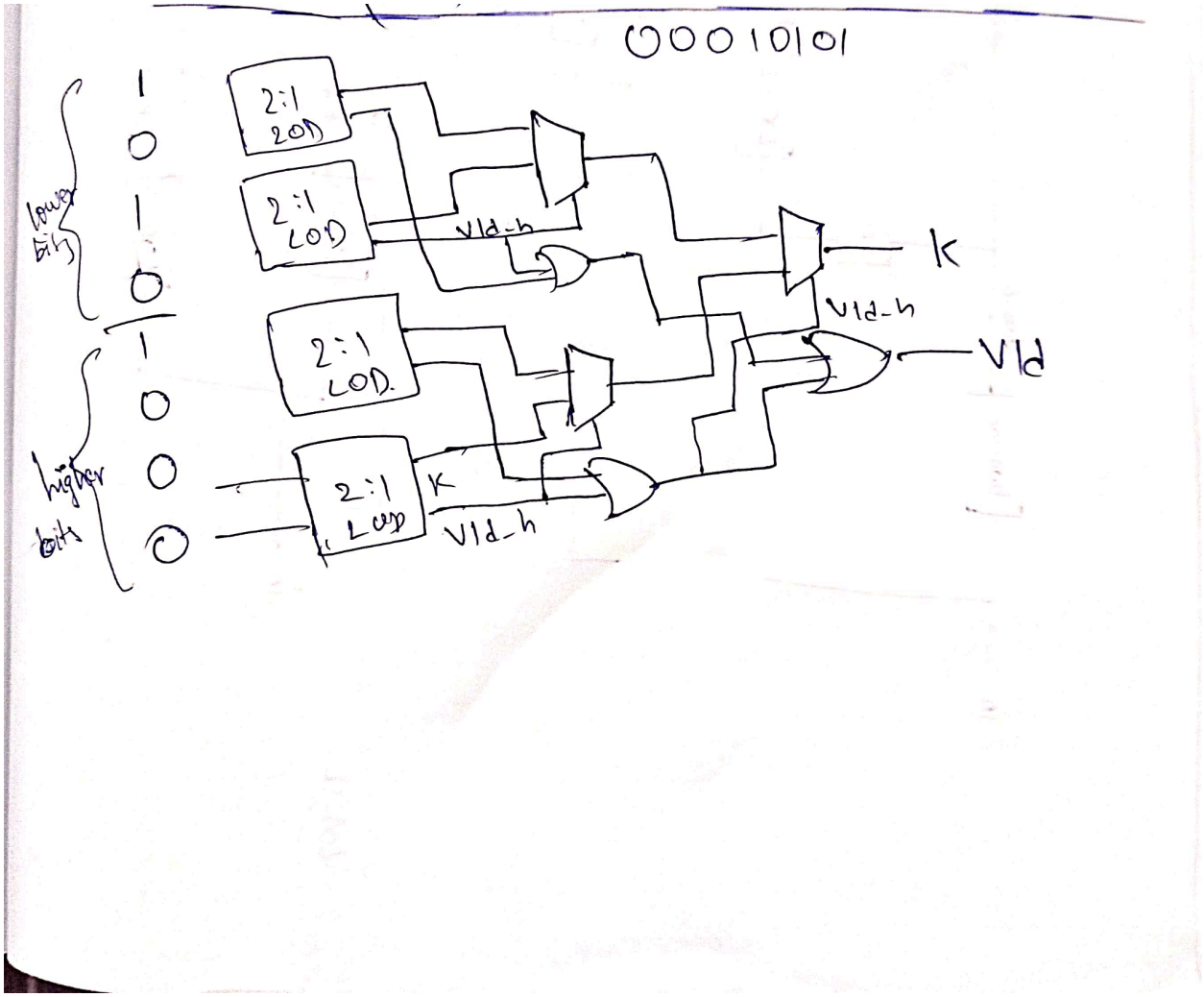
∴ $r = 10_2 \quad e = 1_2 \quad f = 0100_2 \quad s = 1_2$

Using the formula - $v = (2^{2^{ES}})^r \cdot 2^e \cdot (1+f) \times (-1)^s$

$= (2^{2^1})^2 \cdot 2^1 \cdot (1+0.25) \times (-1)^1$

$= 2^4 \cdot 2^1 \cdot (1.25) \times (-1)$

$= -40_{10}$



Posit Data Extraction with Leading Zero's Detection Algorithm

- Let the predetermined values be N - size of the binary string (IN) and ES - the number of exponent bits.
- Now, the first bit has to be the sign bit.

- Let s = IN[N-1]. Now, if s is 0 it denotes a positive sign else a negative sign.
- If all the bits of IN are zero, then it denotes zero (z = NOR(IN[N-1:0]).
- If the sign bit of IN is 1 and the rest of the bits are zero, then it denotes infinity (±∞ as mentioned in “Special cases” section) (inf = IN[N-1] & NOR(IN[N-2:0]).
- If s is 0, then let XIN be IN[N-2:0] (where “XIN” is a temp variable used to store “IN” without sign bit).
- If s is 1, we need to take the 2’s complement of the input string after the sign bit. So, XIN = ~IN[N-2:0] + 1.
- Now, we are using Leading Zero’s Detection Algorithm. So, we need to check for zeroes in the binary string. If XIN[N-2] is 1, then LIN = XIN (where “LIN” is temp variable used to calculate position of leading one).
- If XIN[N-2] is 0, then we need to invert the bits. So, LIN = ~XIN.
- Then, we pass this LIN to the Leading Zero Detector Circuit which gives us the index of the first zero in the input from MSB side, equal to K.
- If the first bit of XIN is 0, then regime value –> r = K-1.
- If the first bit of XIN is 1, then the regime value is 2’s complement of K – r = ~(K-1).
- Let’s left shift XIN by K+1 bits and assign this value to temporary variable temp. So, temp = XIN<<(K+1). This removes all the regime bits and leaves the exponent and fractional bits.
- Now, if N – 2 – K is greater than ES, then the next ES bits are the expo bits, else whatever remaining bits are the expo bits with no fractional bits. So, e = the highest ES bits of temp from the MSB side.
- Now, if we remove the exponent bits by left shifting by ES bits, the remaining bits are the fractional bits. So, f = temp<<ES.
- Now, since we get the r, e and f values, we can calculate the decimal value of the number using the formula below.

$$\left(2^{2^{es}}\right)^{r} \times 2^{e} \times 1.f.$$

In LZD, the algorithm is the same as LOD for finding K, with only one change in the 2:1 LZD, where we will be making a circuit to detect the occurrence of zero and its index in those 2 bits. The OR is replaced by NAND, and the 1stbit is negated instead of 2nd one.

Example LZD Decoding process

LZD

$g_n = 1000\ 1011_2$

$s = 1$

⇒ we need to take 2's complement

⇒ $x_{in} = \overline{0001011} + 1 = 1110100 + 1 = 1110101_2$

First bit of x_{in} is 1

⇒ we can take $l_{in} = x_{in}$.

⇒ $l_{in} = 1110101_2$

⇒ $K = 3$ (as 0 is at the 3rd index)

First bit of x_{in} is 1

⇒ $r = K - 1 = 2 = 10_2$

temp = $1110101 << 4 = 1010000_2$

Here, since $N - K - 2 = 3 > ES = 1$

⇒ e = Highest ES bits of temp

$e = temp[6] = 1_2$

$f = 1010000_2 << 1 = 0100000_2$

$\therefore r = 10_2 \quad e = 1_2 \quad f = 0100_2 \quad s = 1_2$

Using the formula $-V = (2^{2^{es}})^r \cdot 2^e \cdot (1 \cdot f) \cdot (-1)^s$

$= (2^{2^1})^2 \cdot 2^1 \cdot (1 + 0.25) \cdot (-1)^1$

$= 2^4 \cdot 2^1 \cdot (1.25) \times -1$

$= -40$

Posit Data Extraction with Leading Difference Detection Algorithm

- Let the predetermined values be N - size of the binary string (IN) and ES - the number of exponent bits.
- Now, the first bit has to be the sign bit.
- Let s = IN[N-1]. Now, if s is 0 it denotes a positive sign else a negative sign.
- If s is 0, then let XIN be IN[N-2:0] (where “XIN” is a temp variable used to store “IN” without sign bit).
- If s is 1, we need to take the 2’s complement of the input string after the sign bit. So, XIN = ~IN[N-2:0] + 1.
- Now, we need to use the **Leading Difference Detection Algorithm.**
- First, we use XNOR to find if there is any difference(dif bits array, dif[N-2:0]) between two adjacent bits.
- Then, we generate an array of bits known as en(en[N-3:0]). This is found by taking the ‘and’ of all bits from the i-th bit(i goes from LSB to MSB) to MSB.
- Finally, LDD[i] is generated by performing the operation: ~dif[i]&en[i+1].
    - dif[i] is 0 at positions where the ith and (i+1)th bits are different.
    - en[i] is the cumulative AND of all preceding values in dif, this value turns 1 only when all bits preceding the current bit is 1, else it stays 0.
    - By combining ~dif[i] (indicating no difference at position i) with en[i+1] (ensuring all bits remain unchanged up to position i+1), LDD[i] is set to 1 precisely at the first-bit change and remains 0 for all other positions.
- We use these LDD bits as a select lines/bit shifter to shift the input XIN (over all possible combinations) after ignoring the first two bits. We ignore the first two bits because they always need to be flushed without depending on other bits.
- The position of ‘1’ in the LDD result tells how many bits we need to shift the XIN.
- Once we do bit shifting, the first ‘es’ bits give us the exponent value, and the remaining bits are fraction bits.
- We obtain ‘r’ by using LDD as a select line overall pre-defined pairs.
- The bit-shifting of the result is done by the tree of **NAND** gates using LDD as a selection variable.
- The sign bit is the MSB of the input.
- We use the same formula to calculate the decimal representation of the number

$$\left(2^{2^{es}}\right)^{r} \times 2^{e} \times 1.f.$$

Example LDD Decoding process

LDD

In = 1 0 0 0 1 0 1 1

Sign bit.

As sign bit is 1 (number is -ve), we need to take 2's complement.

$x_{in} = \overline{0001011} + 1 = 1110101_2$

diff = XNOR of 2 adjacent bits

diff[i] = 1 if 2 adjacent bits are same,
else 0,

x_{in} = 1 1 1 0 1 0 1

diff = 1 1 0 0 0 0

en[i] = AND of all bits prior to ith bits (MSB's)

en = 1 0 0 0 0

LDD[N-3] = ~ diff[N-3] = 0

LDD[N-4] = diff[N-3] & ~ diff[N-4] = 0

LDD[i] = ~ diff[i] & en[i+1]

LDD = 0 0 1 0 0 0

all possible combinations of LDD are predefined to get regime value.

r' = 2.

$x_{in}[N-2] = 1 \Rightarrow r = r'$

r = 2.

Bitshifting

The minimum bits needed to be flushed are 2 in any case. While taking all possible combinations we ignore first 2 bits of xin.

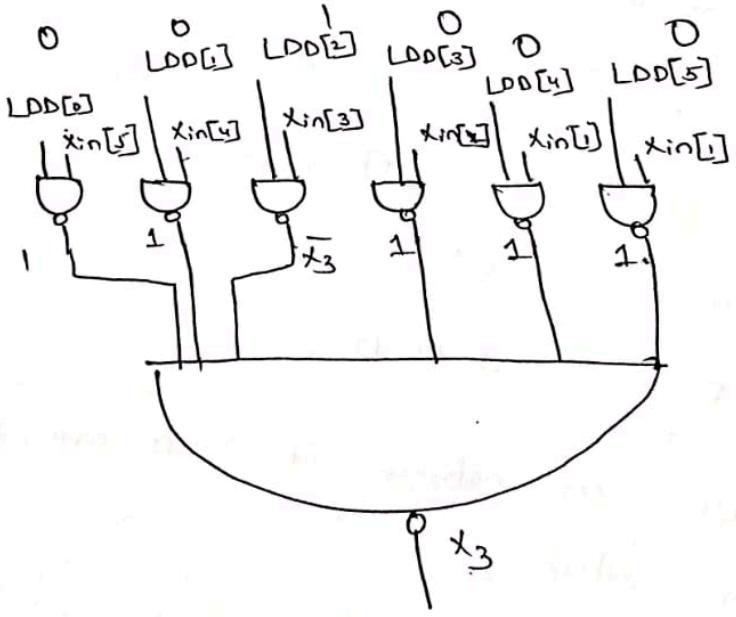
$x_{in} = 1\underline{1}10101$

Ignored.

$\overline{x_{in}} = 10101$

We will append zeroes on right side whereever needed while we shift.

out[5] → To get this,



out[5] → shifted by 2 bits.

Similarly we find remaining out values.

e.g. = out[N-3 : N-3-es+1]

out = 1 0 1 0 0 0

e = 1 bit, frac = 5 bits

frac = [out[n-4 : 0]]

frac = 01000

e = 1

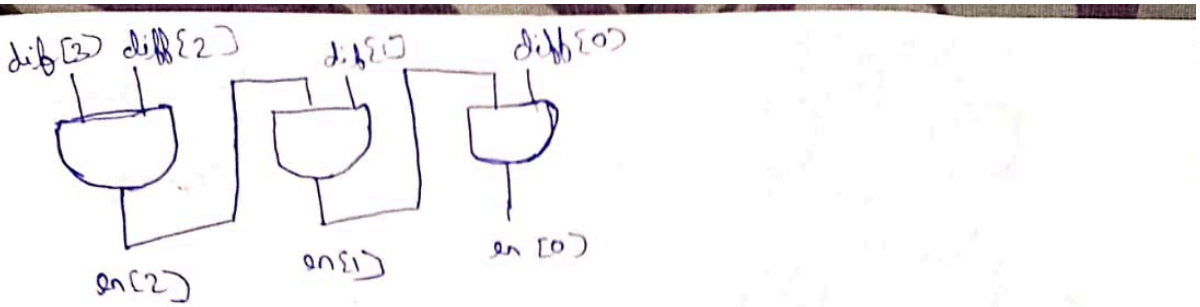
The number is $\left(2^{2^e}\right)^r \cdot 2^e (1+f)(-1)^{sign}$

$= \left(2^{2^1}\right)^2 \cdot 2^1 (1+0.25)(-1)^1$

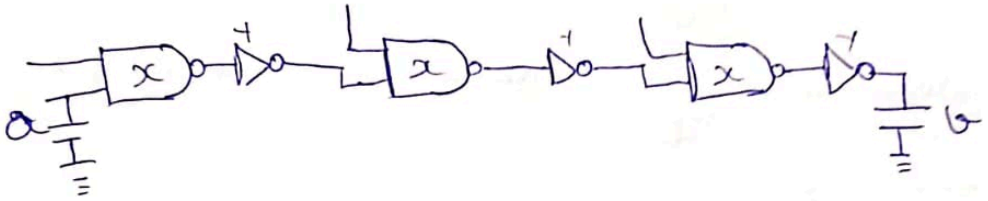
$= (16 \times 2)(1.25)(-1)$

$= -40$

Linear Delay Model for Optimal Structure Estimation



⇒ The above circuit can be redrawn as: (in terms of basic gates)



$P_1=2$ $P_2=1$ $P_3=2$ $P_4=1$ $P_5=2$ $P_6=1$

$g_1=4/3$ $g_2=1$ $g_3=4/3$ $g_4=1$ $g_5=4/3$ $g_6=1$

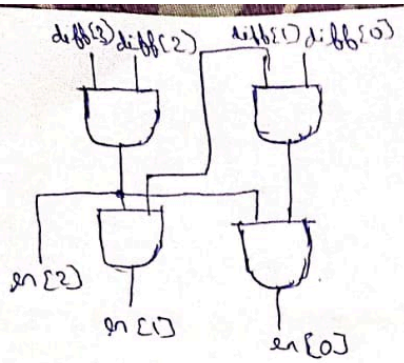
$h_1=\frac{y}{x}$ $h_2=x/y$ $h_3=y/x$ $h_4=x/y$ $h_5=y/x$ $h_6=b/y$

We use linear delay to find the normalized critical path delay:

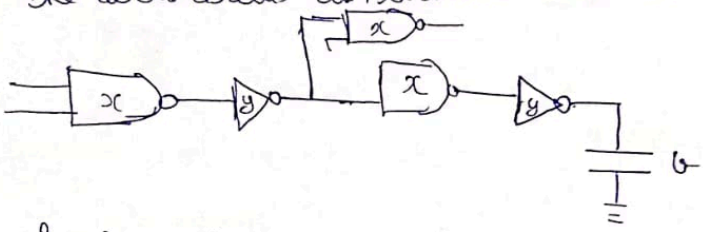
$$Delay_1 = \sum p_i + g_i h_i$$

$$Delay_1 = 9 + \frac{4y}{3x} + \frac{x}{y} + \frac{4y}{3x} + \frac{x}{y} + \frac{4y}{3x} + \frac{b}{y}$$

$$Delay_1 = 9 + 4\frac{y}{x} + \frac{2x+b}{y}$$



The above circuit can be written as : (in terms of basic gates):



$p_1 = 2$ $p_2 = 1$ $p_3 = 2$ $p_4 = 1$

$g_1 = 4/3$ $g_2 = 1$ $g_3 = 4/3$ $g_4 = 1$

$h_1 = y/x$ $h_2 = \frac{2x}{y}$ $h_3 = y/x$ $h_4 = 6/y$

Linear Delay is used to calculate the normalized critical path delay:

$$Delay_2 = \sum p_i + g_i h_i$$

$$Delay_2 = 6 + \frac{4y}{3x} + \frac{2x}{y} + \frac{4y}{3x} + \frac{6}{y}$$

$$Delay_2 = 6 + \frac{8y}{3x} + \frac{2x+6}{y}$$

Comparing $delay_1$ and $delay_2$

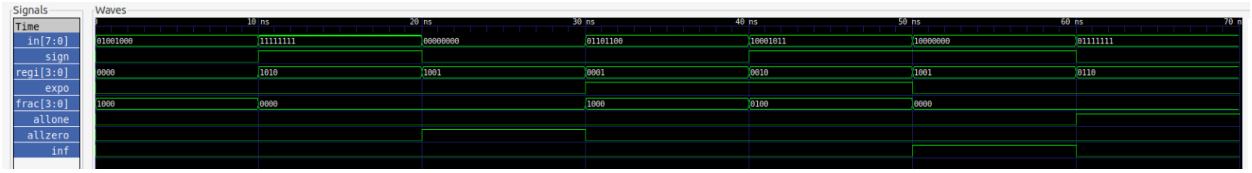
$$Delay_1 - Delay_2 = 3 + \frac{4y}{3x}$$

From the above, we can say that circuit2 is better optimized compared to circuit 1.

LOD Algorithm post-synthesis simulation outputs

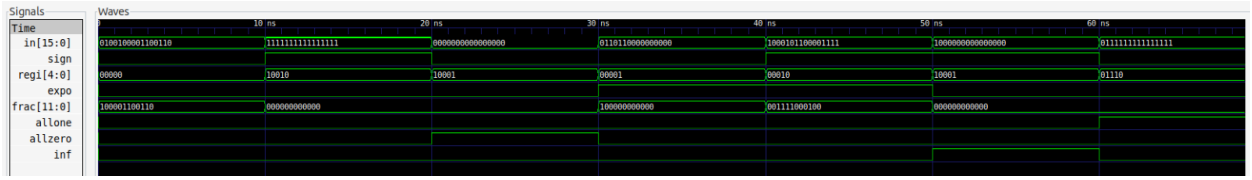
1. 8-bit Posit Decoding using LOD for ES = 1.

```
VCD info: dumpfile decoder_tb_lod_8_1.vcd opened for output.
in = 01001000 | sign = 0 | regi = 0000 | expo = 0 | frac = 1000 | allone = 0 | allzero = 0 | inf = 0
in = 11111111 | sign = 1 | regi = 1010 | expo = 0 | frac = 0000 | allone = 0 | allzero = 0 | inf = 0
in = 00000000 | sign = 0 | regi = 1001 | expo = 0 | frac = 0000 | allone = 0 | allzero = 1 | inf = 0
in = 01101100 | sign = 0 | regi = 0001 | expo = 1 | frac = 1000 | allone = 0 | allzero = 0 | inf = 0
in = 10001011 | sign = 1 | regi = 0010 | expo = 1 | frac = 0100 | allone = 0 | allzero = 0 | inf = 0
in = 10000000 | sign = 1 | regi = 1001 | expo = 0 | frac = 0000 | allone = 0 | allzero = 0 | inf = 1
in = 01111111 | sign = 0 | regi = 0110 | expo = 0 | frac = 0000 | allone = 1 | allzero = 0 | inf = 0
```



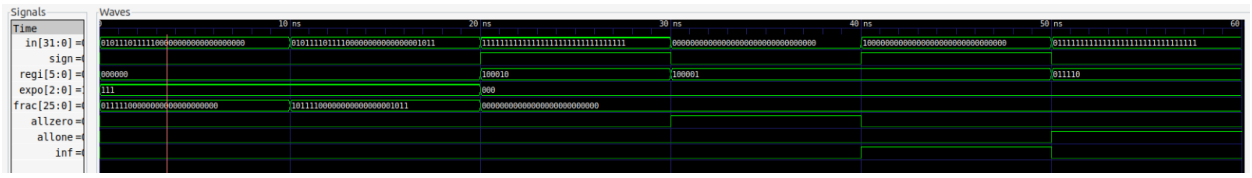
2. 16-bit Posit Decoding using LOD for ES = 1.

```
VCD info: dumpfile decoder_tb_lod_16_1.vcd opened for output.
in = 0100100001100110 | sign = 0 | regi = 00000 | expo = 0 | frac = 1000011100110 | allone = 0 | allzero = 0 | inf = 0
in = 1111111111111111 | sign = 1 | regi = 10010 | expo = 0 | frac = 0000000000000 | allone = 0 | allzero = 0 | inf = 0
in = 0000000000000000 | sign = 0 | regi = 10001 | expo = 0 | frac = 0000000000000 | allone = 0 | allzero = 1 | inf = 0
in = 0110110000000000 | sign = 0 | regi = 00001 | expo = 1 | frac = 1000000000000 | allone = 0 | allzero = 0 | inf = 0
in = 1000101100001111 | sign = 1 | regi = 00010 | expo = 1 | frac = 0011110000100 | allone = 0 | allzero = 0 | inf = 0
in = 1000000000000000 | sign = 1 | regi = 10001 | expo = 0 | frac = 0000000000000 | allone = 0 | allzero = 0 | inf = 1
in = 0111111111111111 | sign = 0 | regi = 01110 | expo = 0 | frac = 0000000000000 | allone = 1 | allzero = 0 | inf = 0
```

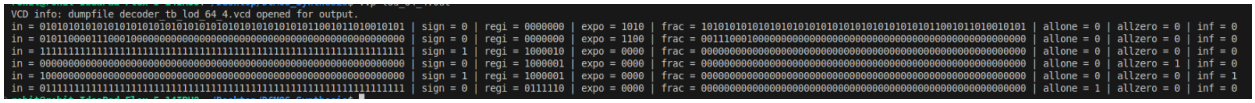


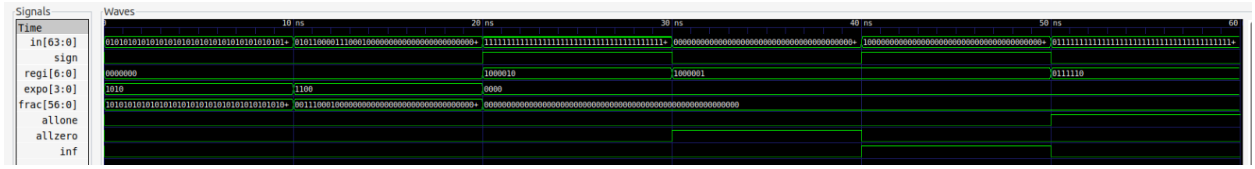
3. 32-bit Posit Decoding using LOD for ES = 3.

```
VCD info: dumpfile decoder_tb_lod_32_3.vcd opened for output.
in = 01011101111100000000000000000000 | sign = 0 | regi = 000000 | expo = 111 | frac = 01111100000000000000000000 | allone = 0 | allzero = 0 | inf = 0
in = 01011101111100000000000000001011 | sign = 0 | regi = 000000 | expo = 111 | frac = 10111100000000000000001011 | allone = 0 | allzero = 0 | inf = 0
in = 11111111111111111111111111111111 | sign = 1 | regi = 100010 | expo = 000 | frac = 00000000000000000000000000 | allone = 0 | allzero = 0 | inf = 0
in = 00000000000000000000000000000000 | sign = 0 | regi = 100001 | expo = 000 | frac = 00000000000000000000000000 | allone = 0 | allzero = 1 | inf = 0
in = 10000000000000000000000000000000 | sign = 1 | regi = 100001 | expo = 000 | frac = 00000000000000000000000000 | allone = 0 | allzero = 0 | inf = 1
in = 01111111111111111111111111111111 | sign = 0 | regi = 011110 | expo = 000 | frac = 00000000000000000000000000 | allone = 1 | allzero = 0 | inf = 0
```



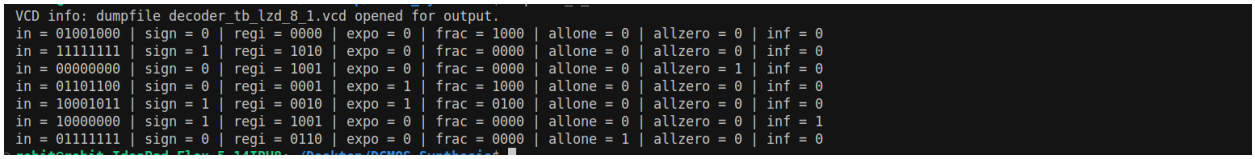
4. 64-bit Posit Decoding using LOD for ES = 4.



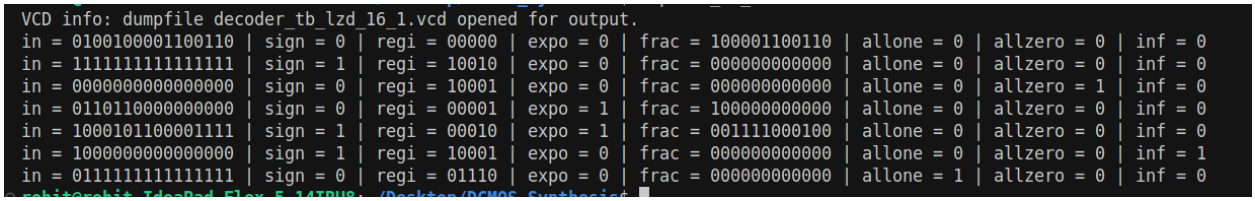


LZD Algorithm post-synthesis simulation outputs

1. 8-bit Posit Decoding using LZD for ES = 1.

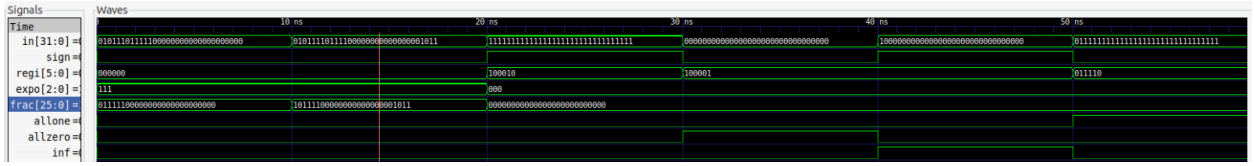


2. 16-bit Posit Decoding using LZD for ES = 1.



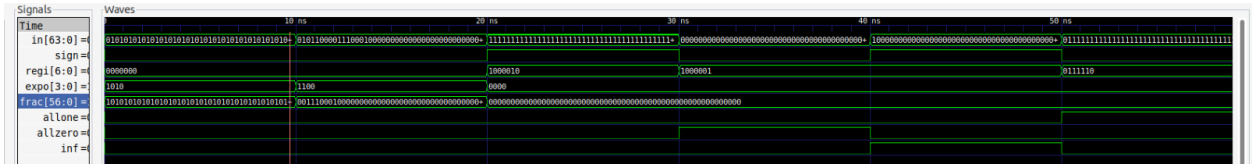
3. 32-bit Posit Decoding using LZD for ES = 3.

```
VCD info: dumpfile decoder_tb_lzd_32_3.vcd opened for output.
in = 01011101111100000000000000000000 | sign = 0 | regi = 000000 | expo = 111 | frac = 01111100000000000000000000 | allone = 0 | allzero = 0 | inf = 0
in = 01011101111100000000000000001011 | sign = 0 | regi = 000000 | expo = 111 | frac = 10111100000000000000001011 | allone = 0 | allzero = 0 | inf = 0
in = 11111111111111111111111111111111 | sign = 1 | regi = 100010 | expo = 000 | frac = 00000000000000000000000000 | allone = 0 | allzero = 0 | inf = 0
in = 00000000000000000000000000000000 | sign = 0 | regi = 100001 | expo = 000 | frac = 00000000000000000000000000 | allone = 0 | allzero = 1 | inf = 0
in = 10000000000000000000000000000000 | sign = 1 | regi = 100001 | expo = 000 | frac = 00000000000000000000000000 | allone = 0 | allzero = 0 | inf = 1
in = 01111111111111111111111111111111 | sign = 0 | regi = 011110 | expo = 000 | frac = 00000000000000000000000000 | allone = 1 | allzero = 0 | inf = 0
```



4. 64-bit Posit Decoding using LZD for ES = 4.

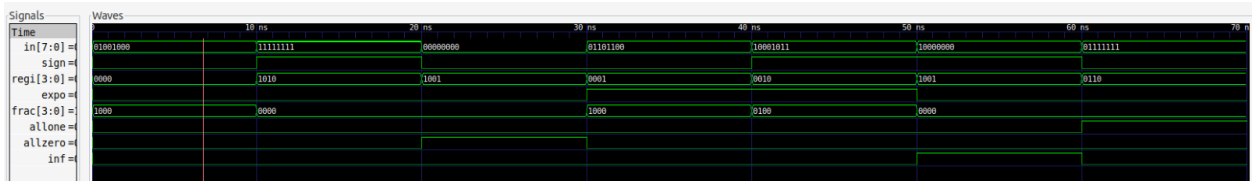
```
VCD info: dumpfile decoder_tb_lzd_64_4.vcd opened for output.
in = 0101010101010101010101010101010101010101010101010110010110010101 | sign = 0 | regi = 0000000 | expo = 1010 | frac = 1010101010101010101010101010101010101011001011010010101 | allone = 0 | allzero = 0 | inf = 0
in = 0101100001110001000000000000000000000000000000000000000000000000 | sign = 0 | regi = 0000000 | expo = 1100 | frac = 0011100010000000000000000000000000000000000000000000000000000000 | allone = 0 | allzero = 0 | inf = 0
in = 1111111111111111111111111111111111111111111111111111111111111111 | sign = 1 | regi = 1000010 | expo = 0000 | frac = 0000000000000000000000000000000000000000000000000000000000000000 | allone = 0 | allzero = 0 | inf = 0
in = 0000000000000000000000000000000000000000000000000000000000000000 | sign = 0 | regi = 1000001 | expo = 0000 | frac = 0000000000000000000000000000000000000000000000000000000000000000 | allone = 0 | allzero = 1 | inf = 0
in = 1000000000000000000000000000000000000000000000000000000000000000 | sign = 1 | regi = 1000001 | expo = 0000 | frac = 0000000000000000000000000000000000000000000000000000000000000000 | allone = 0 | allzero = 0 | inf = 1
in = 0111111111111111111111111111111111111111111111111111111111111111 | sign = 0 | regi = 0111110 | expo = 0000 | frac = 0000000000000000000000000000000000000000000000000000000000000000 | allone = 1 | allzero = 0 | inf = 0
```



LDD Algorithm post-synthesis simulation outputs

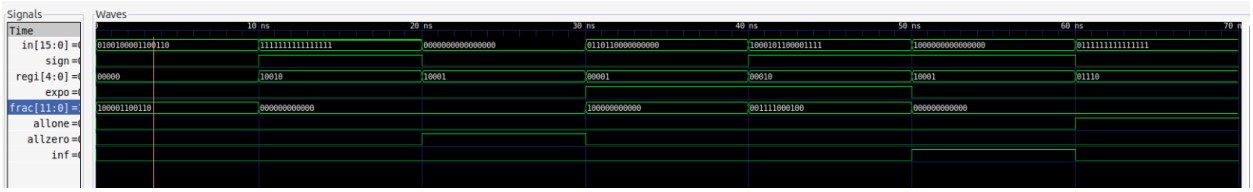
1. 8-bit Posit Decoding using LDD for ES = 1.

```
VCD info: dumpfile decoder_tb_ldd_8_1.vcd opened for output.
in = 01001000 | sign = 0 | regi = 0000 | expo = 0 | frac = 1000 | allone = 0 | allzero = 0 | inf = 0
in = 11111111 | sign = 1 | regi = 1010 | expo = 0 | frac = 0000 | allone = 0 | allzero = 0 | inf = 0
in = 00000000 | sign = 0 | regi = 1001 | expo = 0 | frac = 0000 | allone = 0 | allzero = 1 | inf = 0
in = 01101100 | sign = 0 | regi = 0001 | expo = 1 | frac = 1000 | allone = 0 | allzero = 0 | inf = 0
in = 10001011 | sign = 1 | regi = 0010 | expo = 1 | frac = 0100 | allone = 0 | allzero = 0 | inf = 0
in = 10000000 | sign = 1 | regi = 1001 | expo = 0 | frac = 0000 | allone = 0 | allzero = 0 | inf = 1
in = 01111111 | sign = 0 | regi = 0110 | expo = 0 | frac = 0000 | allone = 1 | allzero = 0 | inf = 0
```



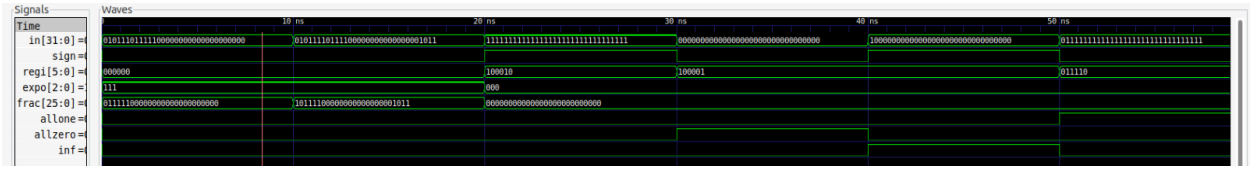
2. 16-bit Posit Decoding using LDD for ES = 1.

```
VCD info: dumpfile decoder_tb_ldd_16_1.vcd opened for output.
in = 0100100001100110 | sign = 0 | regi = 00000 | expo = 0 | frac = 1000011100110 | allone = 0 | allzero = 0 | inf = 0
in = 1111111111111111 | sign = 1 | regi = 10010 | expo = 0 | frac = 0000000000000 | allone = 0 | allzero = 0 | inf = 0
in = 0000000000000000 | sign = 0 | regi = 10001 | expo = 0 | frac = 0000000000000 | allone = 0 | allzero = 1 | inf = 0
in = 0110110000000000 | sign = 0 | regi = 00001 | expo = 1 | frac = 1000000000000 | allone = 0 | allzero = 0 | inf = 0
in = 1000101100001111 | sign = 1 | regi = 00010 | expo = 1 | frac = 0011110001000 | allone = 0 | allzero = 0 | inf = 0
in = 1000000000000000 | sign = 1 | regi = 10001 | expo = 0 | frac = 0000000000000 | allone = 0 | allzero = 0 | inf = 1
in = 0111111111111111 | sign = 0 | regi = 01110 | expo = 0 | frac = 0000000000000 | allone = 1 | allzero = 0 | inf = 0
```



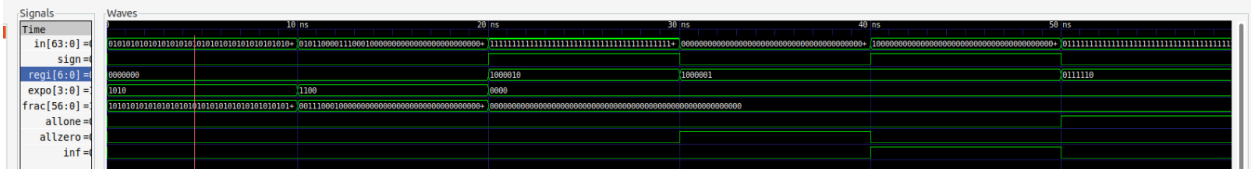
3. 32-bit Posit Decoding using LDD for ES = 3.

```
VCD info: dumpfile decoder_tb_ldd_32_3.vcd opened for output.
in = 01011101111100000000000000000000 | sign = 0 | regi = 000000 | expo = 111 | frac = 01111100000000000000000000 | allone = 0 | allzero = 0 | inf = 0
in = 01011101111100000000000000001011 | sign = 0 | regi = 000000 | expo = 111 | frac = 10111100000000000000001011 | allone = 0 | allzero = 0 | inf = 0
in = 11111111111111111111111111111111 | sign = 1 | regi = 100010 | expo = 000 | frac = 00000000000000000000000000 | allone = 0 | allzero = 0 | inf = 0
in = 00000000000000000000000000000000 | sign = 0 | regi = 100001 | expo = 000 | frac = 00000000000000000000000000 | allone = 0 | allzero = 1 | inf = 0
in = 10000000000000000000000000000000 | sign = 1 | regi = 100001 | expo = 000 | frac = 00000000000000000000000000 | allone = 0 | allzero = 0 | inf = 1
in = 01111111111111111111111111111111 | sign = 0 | regi = 011110 | expo = 000 | frac = 00000000000000000000000000 | allone = 1 | allzero = 0 | inf = 0
```

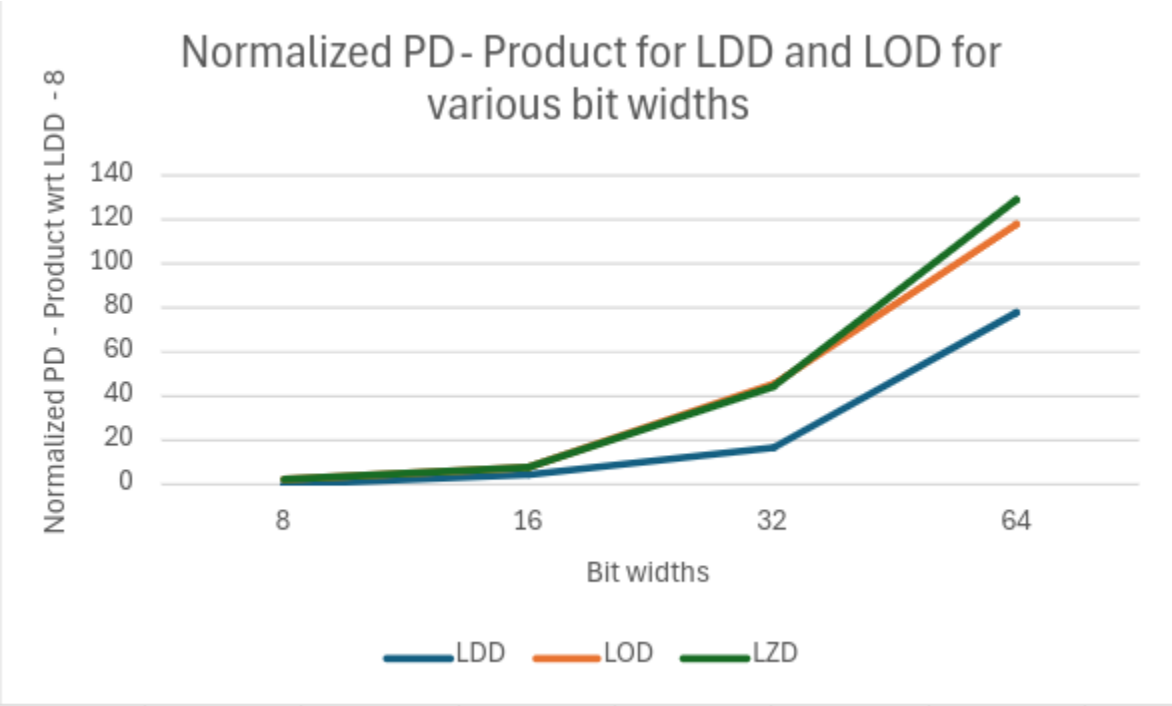


4. 64-bit Posit Decoding using LDD for ES = 4.

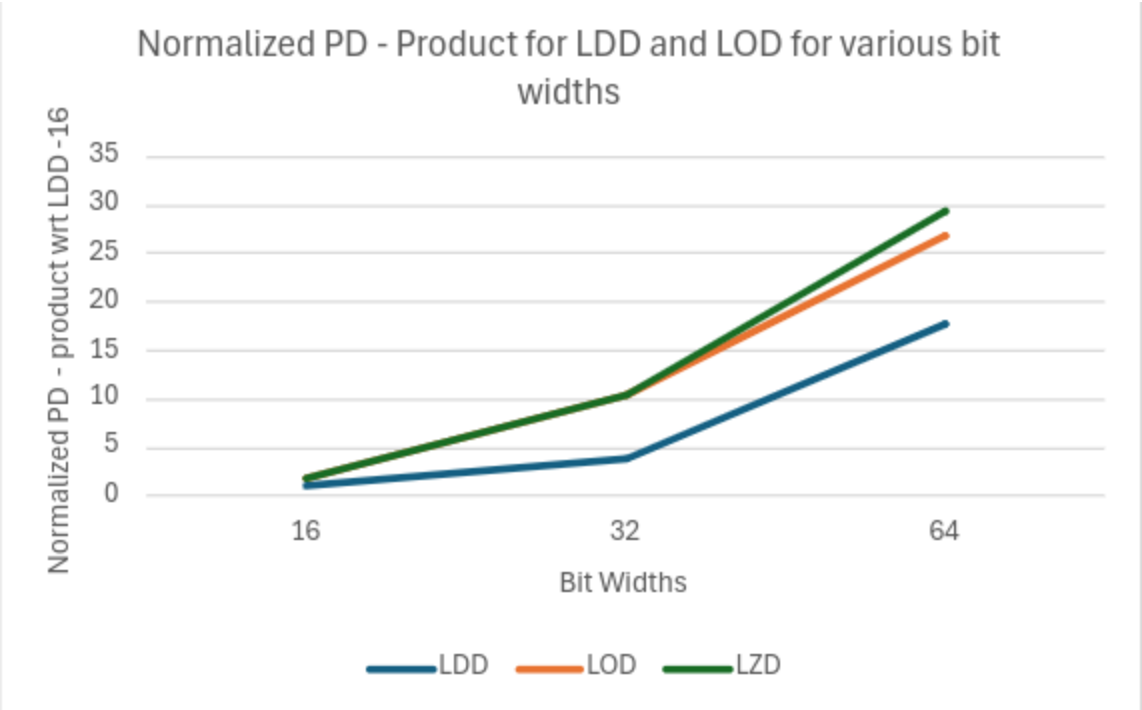
```
VCD info: dumpfile decoder_tb_ldd_64_4.vcd opened for output.
in = 0101010101010101010101010101010101010101010101010101100101100101 | sign = 0 | regi = 0000000 | expo = 1010 | frac = 1010101010101010101010101010101010101100101101001010101 | allone = 0 | allzero = 0 | inf = 0
in = 0101100001110001000000000000000000000000000000000000000000000000 | sign = 0 | regi = 0000000 | expo = 1100 | frac = 0011100010000000000000000000000000000000000000000000000 | allone = 0 | allzero = 0 | inf = 0
in = 1111111111111111111111111111111111111111111111111111111111111111 | sign = 1 | regi = 1000010 | expo = 0000 | frac = 0000000000000000000000000000000000000000000000000000000 | allone = 0 | allzero = 0 | inf = 0
in = 0000000000000000000000000000000000000000000000000000000000000000 | sign = 0 | regi = 1000001 | expo = 0000 | frac = 0000000000000000000000000000000000000000000000000000000 | allone = 0 | allzero = 1 | inf = 0
in = 1000000000000000000000000000000000000000000000000000000000000000 | sign = 1 | regi = 1000001 | expo = 0000 | frac = 0000000000000000000000000000000000000000000000000000000 | allone = 0 | allzero = 0 | inf = 1
in = 0111111111111111111111111111111111111111111111111111111111111111 | sign = 0 | regi = 0111110 | expo = 0000 | frac = 0000000000000000000000000000000000000000000000000000000 | allone = 1 | allzero = 0 | inf = 0
```



Experimental Results:



Normalized power-delay product (energy), with respect to the power-delay product of LDD 8 for various bit widths



Normalized power-delay product (energy), with respect to the power-delay product of LDD 16 for various bit widths

	LDD16	LOD16	LDD32	LOD32	LDD64	LOD64
area (in micro m^2)	304.72	373.81	998.98	1122.10	3375.54	3637.84
power (in micro watt)	3.44	5.47	8.72	14.35	22.56	28.87
delay (in nano s)	3.43	3.82	5.35	8.69	9.26	10.99
pd-product (in fJ)	11.79	20.89	46.66	124.63	208.92	317.21
normalized pd-product	1.00	1.77	3.96	10.57	17.73	26.91

Table of values of area, power, delay, power-delay product and normalized power-delay product for several LOD and LDD wrt LDD 16.