PARAMETERIZED DISTRIBUTED ARITHMETIC FIR FILTER

FIR FILTER

FIR Filter equation: $y[n]=\sum_{k=0}^{N-1}h[k]x[n-k]$ (1)

Figure 1 shows a conventional 4-tap FIR filter architecture. In this case, the $x[k]$ values represent the registers' outputs which are continuously updated:

$y[n]=h[0]x[n]+h[1]x[n-1]+h[2]x[n-2]+h[3]x[n-3]$

For simplicity's sake, Eq. 1 is rewritten as: $y=\sum_{k=0}^{N-1}h[k]x[k]$.

We have omitted the index of $y[n]$; it is understood that a new value y is obtained at each clock cycle. Fig. 2 shows the filter with these updates for N=4. The equation results:

$y=h[0]x[0]+h[1]x[1]+h[2]x[2]+h[3]x[3]$

DANIEL LLAMOCCA © ivpcl@unm

A linear phase response FIR Filter has symmetric coefficients around the center values. So we can add the symmetric taps before they are multiplied by the coefficients. Fig. 3 shows how the symmetry halves the number of multiplications from 8 to 4, this reduces the circuitry required to implement the filter. The filter equation can be rewritten as:

$$y=h[0]s[0]+h[1]s[1]+h[2]s[2]+h[3]s[3]$$

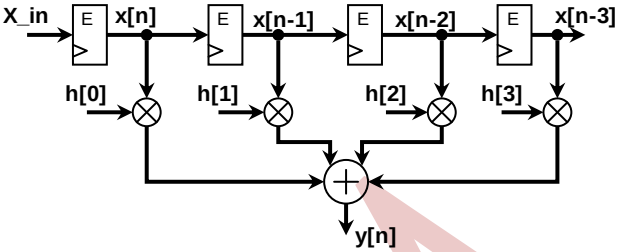

Figure 1. 4-tap FIR filter architecture

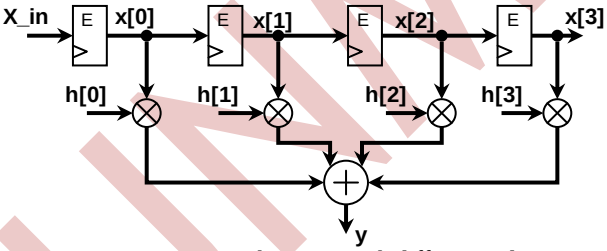

Figure 2. 4-tap FIR architecture with different indices

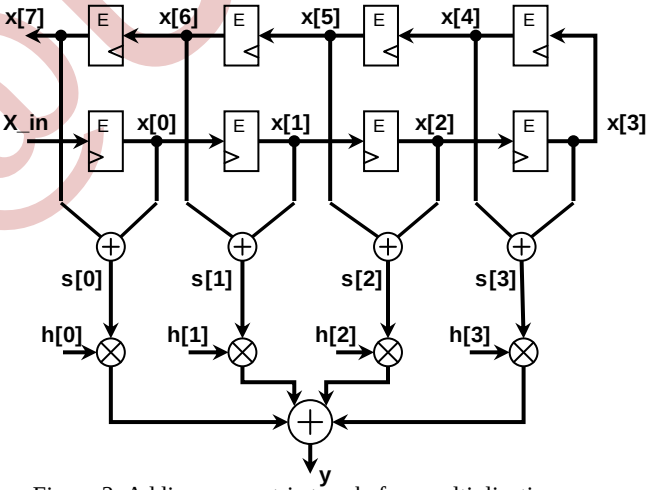

Figure 3. Adding symmetric taps before multiplication

DISTRIBUTED ARITHMETIC TECHNIQUE:

For simplicity's sake, we'll use the non-symmetric filter. Symmetric filter results can be easily derived from the derivations in this section, and more importantly, they are explained in the next document, though for a specific implementation.

Recall: $y=\sum_{k=0}^{N-1}h[k]x[k]$ ← also called 'inner product y'

If the coefficients $h[k]$ are known a priori, then the partial product term $h[k]x[k]$ becomes a multiplication with a constant. This characteristic makes it possible the use of the Distributed Arithmetic Technique. To understand this paradigm, let's start by unfolding the FIR Equation:

$$y=\sum_{k=0}^{N-1}h[k]x[k]=h[0]x[0]+h[1]x[1]+h[2]x[2]+\ldots+h[N-1]x[N-1]$$

UNSIGNED DA SYSTEM: $x[k]$ is represented by: $x[k]=\sum_{b=0}^{B-1}x_b[k]\times 2^b$, $x_b[k]\in\{0,1\}$

Where $x_b[k]$ denotes the b[th] bit of $x[k]$ (which has 'B' bits). Then: $y=\sum_{k=0}^{N-1}\left(h[k]\times\sum_{b=0}^{B-1}x_b[k]2^b\right)$

$$\begin{aligned}y=\;&h[0]\left(x_{B-1}[0]2^{B-1}+x_{B-2}[0]2^{B-2}+\ldots+x_0[0]2^0\right)+\\&h[1]\left(x_{B-1}[1]2^{B-1}+x_{B-2}[1]2^{B-2}+\ldots+x_0[1]2^0\right)+\\&\vdots\\&h[N-1]\left(x_{B-1}[N-1]2^{B-1}+x_{B-2}[N-1]2^{B-2}+\ldots+x_0[N-1]2^0\right)\end{aligned}$$

Now, the summation is redistributed as follows:

$$y = (h[0]x_{B-1}[0] + h[1]x_{B-1}[1] + \ldots + h[N-1]x_{B-1}[N-1]) \times 2^{B-1} +$$
$$(h[0]x_{B-2}[0] + h[1]x_{B-2}[1] + \ldots + h[N-1]x_{B-2}[N-1]) \times 2^{B-2} +$$
$$\vdots$$
$$(h[0]x_0[0] + h[1]x_0[1] + \ldots + h[N-1]x_0[N-1]) \times 2^0$$

$$y = \sum_{b=0}^{B-1}\left(2^b \times \sum_{k=0}^{N-1} h[k]x_b[k]\right) = \sum_{b=0}^{B-1}\left(2^b \times \sum_{k=0}^{N-1} h[k]x_b[k]\right) = \sum_{b=0}^{B-1} 2^b \times f(h[k], x_b[k])$$

$$f(h[k], x_b[k]) = \sum_{k=0}^{N-1} h[k]x_b[k], \quad x_b = [x_b[0] \quad x_b[1] \quad \ldots \quad x_b[N-1]]$$

Preferred implementation of $f(h[k], x_b[k])$: A 2^N-word LUT preprogrammed to accept an N-bit input vector $x_b = [x_b[0] \quad x_b[1] \quad \ldots \quad x_b[N-1]]$ and output $f(h[k], x_b[k])$. Then, each $f(h[k], x_b[k])$ is weighted by 2^b and finally all of them are accumulated.

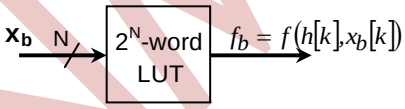

Figure 4. 2^N-word LUT

SIGNED DA SYTEM:

2's complement: MSB tells the sign. Thus, we use the following (B+1)-bit representation:

$$x[k] = -2^B x_B[k] + \sum_{b=0}^{B-1} x_b[k] \times 2^b$$

The inner product y, now becomes: $y = \sum_{k=0}^{N-1}\left(h[k] \times \left(-2^B x_B[k] + \sum_{b=0}^{B-1} x_b[k]2^b\right)\right)$

Using a similar procedure as in the previous case, the inner product results:

$$y = -2^B \sum_{k=0}^{N-1} h[k]x_B[k] + \sum_{b=0}^{B-1}\left(2^b \sum_{k=0}^{N-1} h[k]x_b[k]\right) = -2^B f(h[k], x_B[k]) + \sum_{b=0}^{B-1} 2^b f(h[k], x_b[k])$$

The new term requires a slight implementation modification. There are basically 2 ways:

- Modify the LUT: When $b = B$, it outputs $-f(h[k], x_B[k])$. Then, multiply each term by 2^b and add them all.
- Do not modify the LUT: When $b = B$, it outputs $f(h[k], x_B[k])$. Then, multiply each term by 2^b. We have to account for the negative sign when adding all the terms, e.g. an accumulator with add/subtract control.

There are basically 2 paradigms to implement the inner product in its new form.

i) Iterative Implementation: We make use of a shift-adder as shown in Fig. 5.
   - A vector x_b is fed into the 2^N-word LUT at each clock cycle.
   - Instead of shifting each intermediate value $f(h[k], x_b[k])$ by 'b' bits (which demands an expensive barrel shifter), it is more appropiate to shift the accumulator content itself in each iteration one bit to the right.
   - The adder unit includes a add/sub control so that when $b = B$, it will subtract the $f(h[k], x_B[k])$ from the current result.

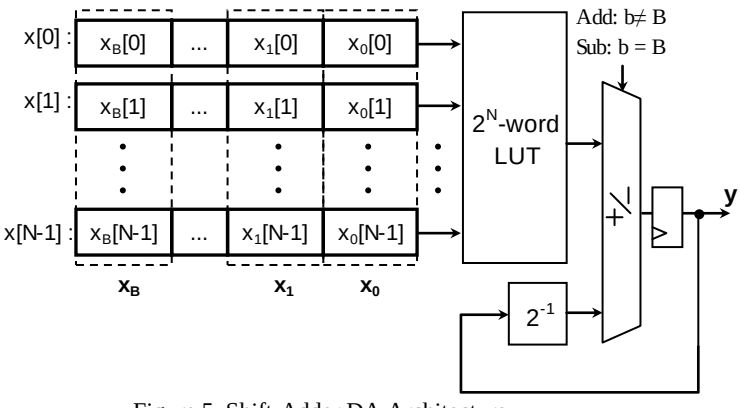

Figure 5. Shift-Adder DA Architecture

   - The shift-adder implementation requires the use of N shift registers of 'B+1' length.

ii) Fully parallel implementation: We use an array of 2^N-word LUTs as shown in Figure 6.
- There are no shift registers here.
- Each of the vectors x_b is fed to a 2^N-word LUT. As a result, we use $B+1$ 2^N-word LUTs.
- The output of each 2^N-word LUT is multiplied by its correspondent 2^b.
- To account for the negative sign in $f(h[k],x_B[k])$, we multiply it by -2^B. Another option is to modify the LUT so that when $b=B$ it outputs $-f(h[k],x_B[k])$.
- All the LUT outputs are weighted by 2^b and added into a final result.

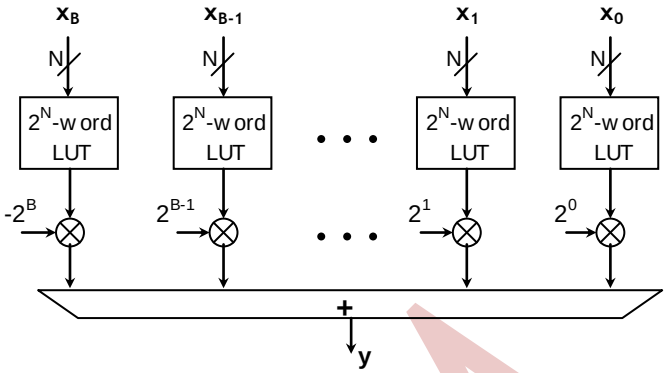

Figure 6. Fully Parallel DA Architecture

MODIFIED DA IMPLEMENTATION

The LUT implementation becomes prohibitively expensive when N is large (if N = 32 → the LUT has 2^{32} words = 4G words!!!). A solution is to divide the filter into filter blocks with L terms, i.e. we have N/L inner products of L terms, as follows:

$$y=\sum_{k=0}^{L-1}h[k]x[k]+\sum_{k=L}^{2L-1}h[k]x[k]+\sum_{k=2L}^{3L-1}h[k]x[k]+\ldots+\sum_{k=\left(N/L-1\right)L}^{N-1}h[k]x[k]$$

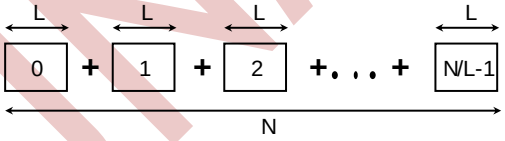

Figure 7. Division of the N coefficients. $N=L^{\circ}$

Each of the N/L summations is transformed to DA form, and then computed in parallel. Finally, we add up al the resulting N/L values. With this in mind, we reformulate the 2 basic implementations:

i) Iterative Implementation: Here we use N/L 2^L-word LUTs. A vector x_b $(0\le b\le B)$ is fed into the LUT at each clock cycle. All LUTs outputs are accumulated; the final result goes through a shift-adder unit. Fig. 8 depicts this implementation. Table 1 illustrates the resource savings attained.

TABLE 1. LUT SPACE COMPARISONS.

Iterative DA implementation	LUT Size	Total space required
No division in filter blocks	2^N words	2^N words
Division into N/L filter blocks	2^L words	$2^L\times N/L$ words

As an example, consider N = 32, L = 4. Then the original DA uses $2^{32}=4G\,words$, while the Modified DA uses $2^4\times\frac{32}{4}=128\,words$. This is a vast improvement at the expense of one extra adder tree.

ii) Fully Parallel Implementation: The output of each of the N/L filter blocks is computed as in the case of Figure 6. The only diference is that the x_b vectors are of 'L' bits; each of these vectors is fed into a 2^L-word LUT (we use $B+1$ 2^L-word LUTs per filter block). Finally the N/L filter block outputs are added in parallel. Figure 9 depicts this implementation. Table 2 illustrates the resource savings attained.

TABLE 2. LUT SPACE COMPARISONS – FULLY PARALLEL IMPLEMENTATION

Implementation	LUT Size	Total space required
No division in filter blocks	$2^N\times(B+1)$ words	$2^N\times(B+1)$ words
Division into N/L filter blocks	$2^L\times(B+1)$words	$2^L\times(B+1)\times N/L$ words

As an example, consider N = 32, L = 4, B = 11. Then the original DA uses $2^{32}\times(11+1)=48G\,words$, while the Modified DA uses $2^4\times(11+1)\times\frac{32}{4}=1536\,words$. This is vast improvement.

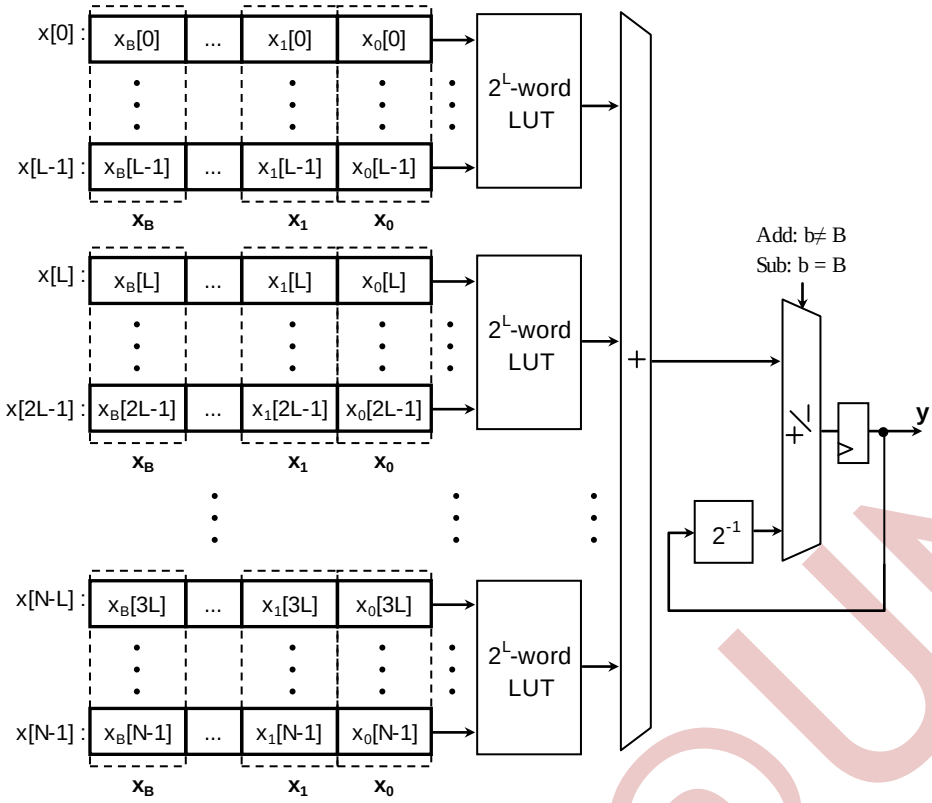

Figure 8. Iterative Modified DA Implementation.

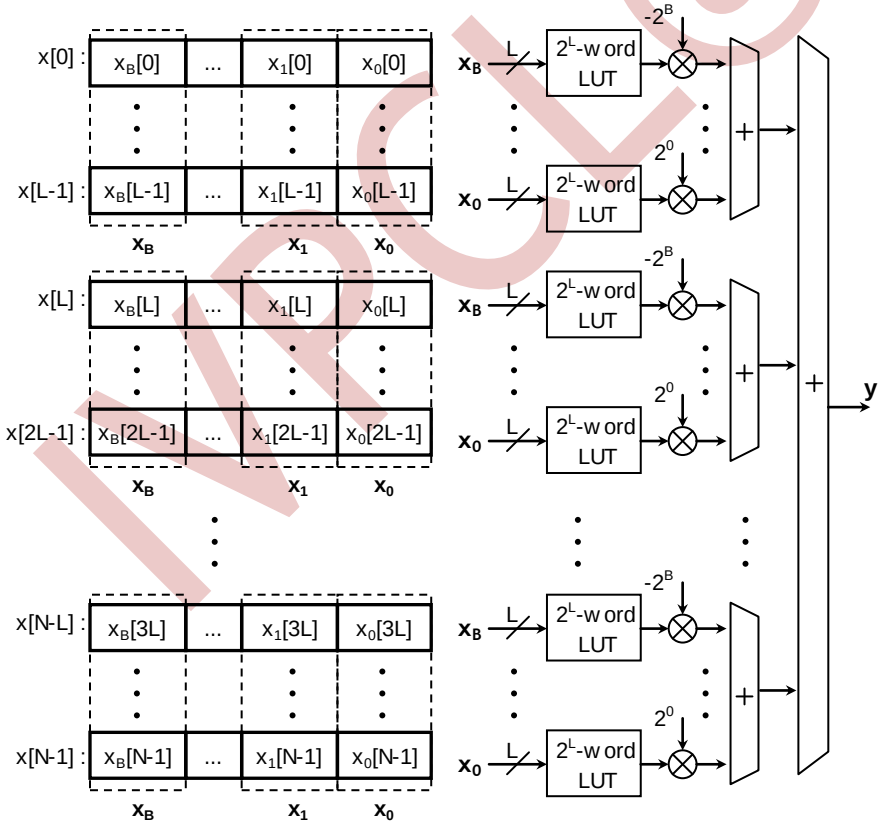

Figure 9. Fully Parallel Modified DA Architecture

A GENERAL DA FILTER IMPLEMENTATION

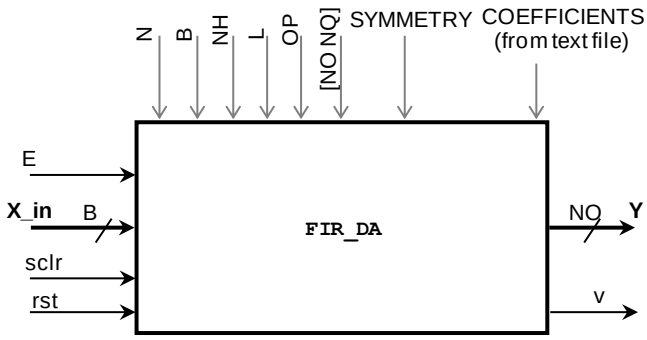

Figure 10. Parameterizable FIR DA module

We describe a FIR Filter core with such a flexibility that allows a friendly interaction with the Xilinx® Partial Reconfiguration process.

This parameterized FIR filter architecture in VHDL allows modification of features such as the number of filter coefficients, the filter coefficients, and the bit precision. We will use the fully parallel modified DA implementation (Fig. 9) with 2's complement representation.

Fig. 10 shows the FIR Filter module with its inputs, outputs, and parameters. Signal 'E' controls the input validity. Clearing the register chain ('sclr' signal) at will is an important requirement when filtering finite size signals. 'v': output valid (a delayed version of 'E', except when E and sclr are asserted. 'rst' clears the shift register for 'v'.

I. FIR FILTER CORE PARAMETERS

- N: Number of taps.
- NH: Number of bits of the coefficient values
- B: Number of bits of the input values
- L: LUT input size. We can choose the input bit-width of the internal LUTs which are the ones that contain the coefficient values' information. $L \geq 2$ (otherwise there is no LUT)
- OP: Output truncation scheme:
  - OP = 0: A truncation is performed at the LSBs, and then saturation is performed, so that the output fixed-point format results [NO NQ].
  - OP = 1: A truncation is performed at both the LSB and MSB parts, so that the output fixed-point format results [NO NQ].
  - OP = 2: The maximum number of allowable bits is used for the output format, which is given by the following formula: $[NH + B - 2 + \lceil \log_2(N+1) \rceil - 1 \quad NH + B - 2]$
- [NO NQ]: Output fixed-point format (valid if OP ≠ 2). NO: number of integer bits. NQ: number of fractional bits. The user can choose the output format, provided is within the bounds of the maximum allowable number of bits for both the integer and fractional part.
- SYMMETRY: Establishes whether or not the FIR filter is symmetric.
  - SYMMETRY = YES: We take advantage of this fact to come up with an improved architecture.
  - SYMMETRY = AYES: Some filters are antisymmetric.
  - SYMMETRY = NO: Some filters (e.g. polyphase filters) are not symmetric.
- COEFFICIENTS: They determine the response of the filter. A text file with the LUT values that represent the filter coefficient is read by the FIR filter VHDL code.

We assume that the inputs/coefficients values are constrained to $[1,1\rangle$. All our formula derivations regarding fixed-point formats of input/output are based on this assumption. Other bounds can always be assumed, but a different derivation for the I/O fixed-point format has to be completed. In addition, there are variables dependent on the parameters that will be used throughout this document; they are defined in Table 3.

TABLE 3. FILTER VARIABLES

Filter type	Number of effective taps *M*	Bit-width of s*[k]* or x*[k]* *sizeI*	Filter equation
Symmetric/ anti-symmetric	$\lceil N/2 \rceil$	$B+1$	$y = \sum_{k=0}^{M-1} h[k]s[k]$
Non-symmetric	N	B	$y = \sum_{k=0}^{N-1} h[k]x[k]$

SYMMETRY CONSIDERATIONS:

Fig. 11 depicts the implementation schemes for both the symmetric and the non-symmetric case. The system is composed of an array of registers and adders (symmetric case) or subtractors (antisymmetric case), a rearrangement stage, an array of Filter Blocks, and a tree adder.

- Symmetric/Anti-symmetric FIR Filter: In this case, we have linear response filters, whose coefficients are symmetric around the center values. This property allows us to add symmetric taps before multiplication in order to reduce the number of multiplications by half, as shown in Figure 3.
  When N is odd, one summation is not performed (the term $s[M-1]$ is not a summation)

TABLE 4. DETAIL OF THE SUMMATIONS FOR BOTH CASES OF N. SYMMETRY → +, ANTISYMMETRY → -

N even: $N = 2M$. There are M summations:	N odd: $N = 2M-1$. There are $M-1$ summations:
$s[0] \leftarrow x[0] \pm x[N-1]$ $s[1] \leftarrow x[1] \pm x[N-2]$ ⋮ $s[i] \leftarrow x[i] \pm x[N-(i+1)]$ ⋮ $s[M-1] \leftarrow x[M-1] \pm x[N-(M-1+1)] = x[M-1] \pm x[M]$	$s[0] \leftarrow x[0] \pm x[N-1]$ $s[1] \leftarrow x[1] \pm x[N-2]$ ⋮ $s[i] \leftarrow x[i] \pm x[N-(i+1)]$ ⋮ $s[M-2] \leftarrow x[M-2] \pm x[N-(M-2+1)] = x[M-2] \pm x[M]$ $s[M-1] \leftarrow x[M-1]$

- Non-symmetric FIR Filter: These filters are useful in many instances, e.g. polyphase filter implementation. Here, there are no summations at all.

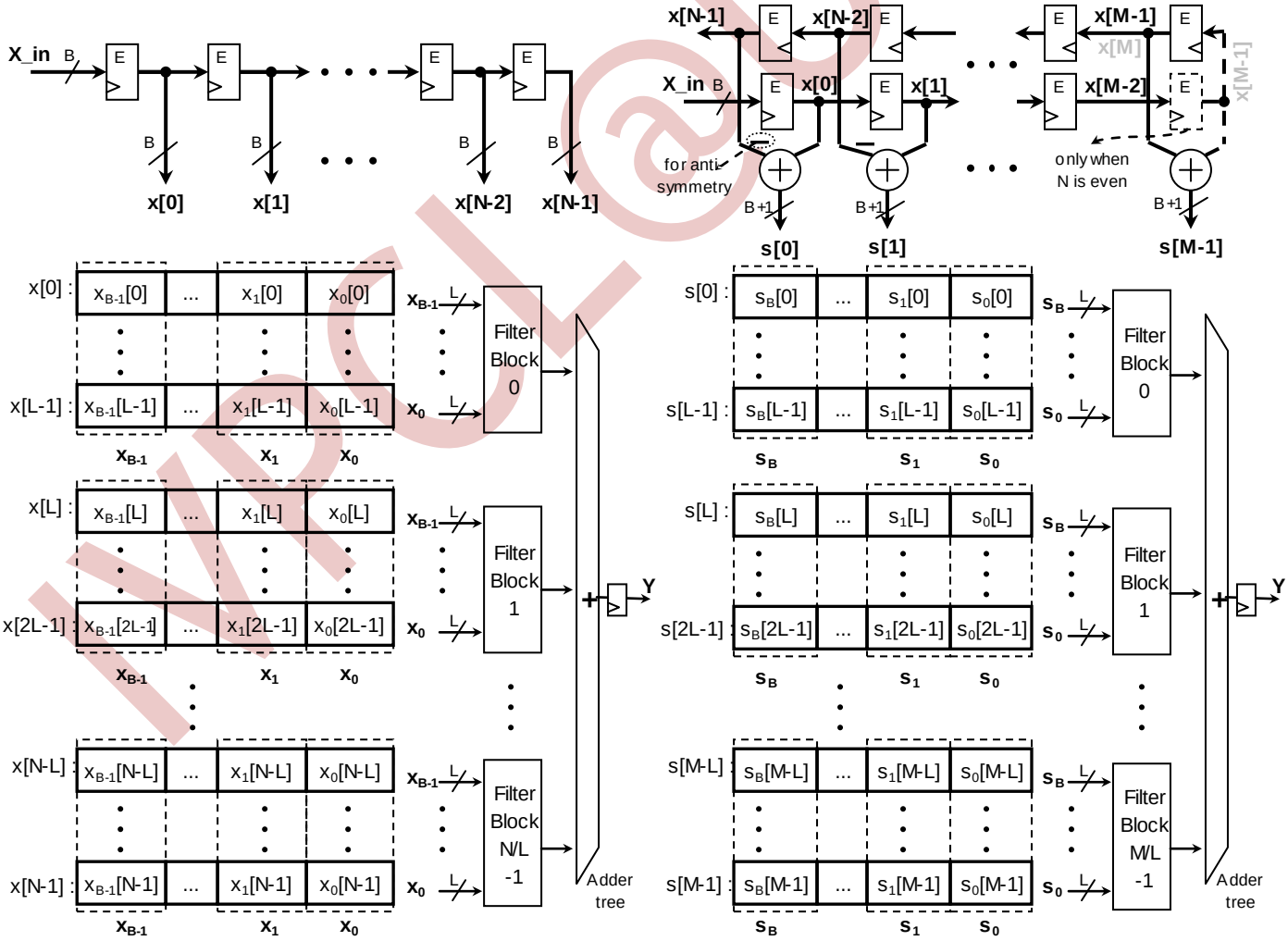

Figure 11. High-performance DA implementation based on the LUT input size (L). Non-symmetric filter (left), Symmetric/Anti-symmetric filter (right)

II. ARRAY OF REGISTERS AND ADDERS

They can be seen at the top part of Fig. 11. They output the $u[k]$ signals, $0 \le k \le M-1$ to the next stage.

- The $x[k]$ bit-width is 'B', i.e. we are using the two's complement B-bits representation.
- The $s[k]$ bit-width is 'B+1', since the summation adds 1 bit. Note that the $s[M-1]$ signal has 'B+1' bits even when there is no summation (N odd). This is to maintain uniformity.

III. FILTER BLOCKS

The distributed arithmetic technique rearranges the input sequence (be it x[k] or s[k]) into vectors of length M, that require an array of $sizeI$ M-input LUTs. This becomes prohibitively expensive if M is large. As a result, for efficient implementation, we divide the filter into M/L filter blocks (see Fig. 11), which are FIR filters that contain a subset of the coefficients. The new filter equation results:

$$y=\sum_{k=0}^{L-1}h[k]u[k]+\sum_{k=L}^{2L-1}h[k]u[k]+\sum_{k=2L}^{3L-1}h[k]u[k]+\ldots+\sum_{k=\left(M/L-1\right)L}^{M-1}h[k]u[k],\quad \begin{matrix} Symmetric\quad filter: u[k]=s[k], MSB=B \\ Non-symmetric\quad filter: u[k]=x[k], MSB=B-1 \end{matrix}$$

Table 5 illustrates the resource savings attained when applying this division technique. Each filter block works with L coefficients, and requires $sizeI$ L-input LUTs (in Fig. 12 each vector of size L goes to an L-input LUT). Each of the M/L filter blocks computes the following equation in the distributed arithmetic fashion:

$$\sum_{k=iL}^{(i+1)L-1}h[k]u[k]=-2^{MSB}\sum_{k=iL}^{(i+1)L-1}h[k]u_{MSB}[k]+\sum_{b=0}^{MSB-1}\left(2^b\sum_{k=iL}^{(i+1)L-1}h[k]u_b[k]\right)=-2^{MSB}f\left(h[k],u_{MSB}[k]\right)+\sum_{b=0}^{MSB-1}2^b f\left(h[k],u_b[k]\right),\quad 0\le i\le M/L-1$$

TABLE 5. LUT SPACE COMPARISONS

Case	Total space required
1 Filter of size M LUTs have M inputs	$sizeI\times 2^M$ words
M/L filter blocks of size L LUTs have L inputs	$sizeI\times 2^L\times M/L$ words

For example, if M = 16, L = 4, then $2^{16} >> 2^4\times 16/4$. Thus, dividing the filter into M/L filter blocks saves a great deal of hardware resources. This realization needs an extra adder tree to add up all filter block outputs. Fig. 12 depicts the i[th] filter block ($0\le i\le M/L-1$) for both symmetric and non-symmetric cases. To account for the negative sign in $x_{B-1}[k]$ (or $s_B[k]$), instead of multiplying the LUT output by -2^{B-1} (or -2^B), we modify the LUT so that it outputs $-f(h[k],x_B[k])$ (or $-f(h[k],s_B[k])$). This is marked by LUT* in Fig. 12.

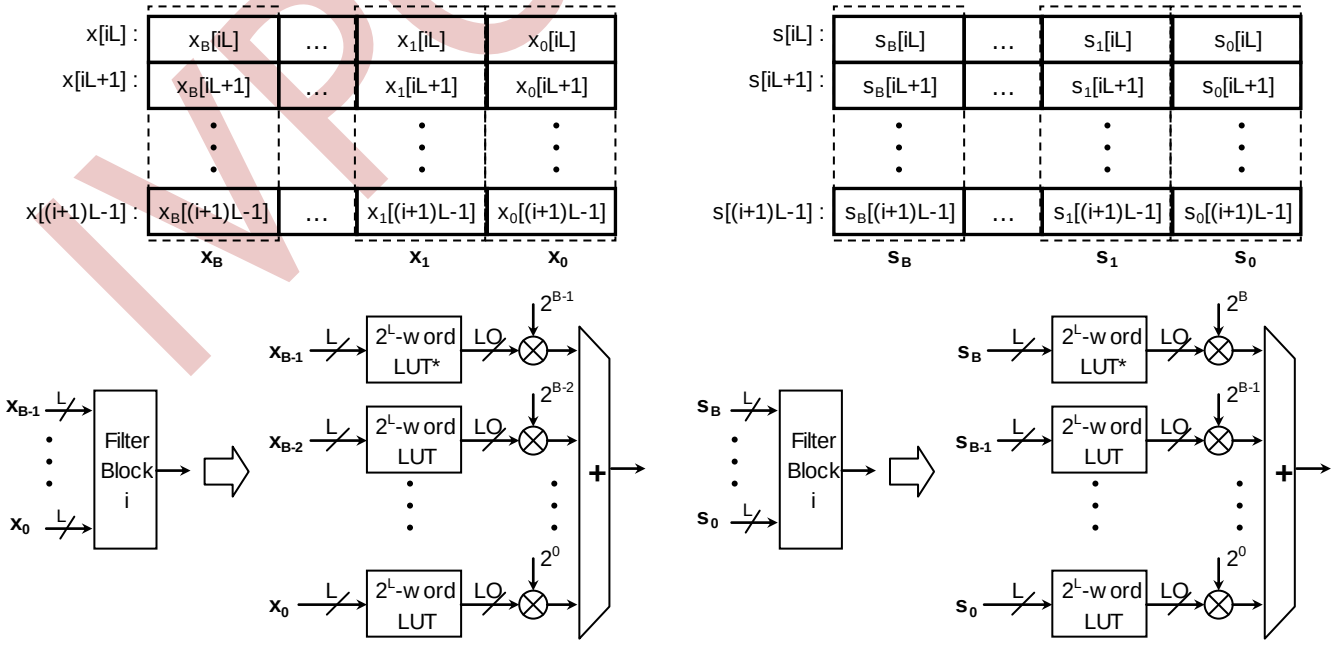

Figure 12. Filter block. Non-symmetric filter block (left). Symmetric filter block (right)

FILTER BLOCK REALIZATION

If we plan to add all the shifters from Figure 12, we will need an adder tree that takes $sizeI$ inputs. As explained in Section IV, that adder tree will require $\lceil log_2(sizeI) \rceil$ adder levels.

However, each summand would require a different and large wordlength, complicating the design. A technique to embed the shifters along the adder tree levels is explained in this section. To this purpose, we begin with an example, where $sizeI = 16$, so the adder tree is full.

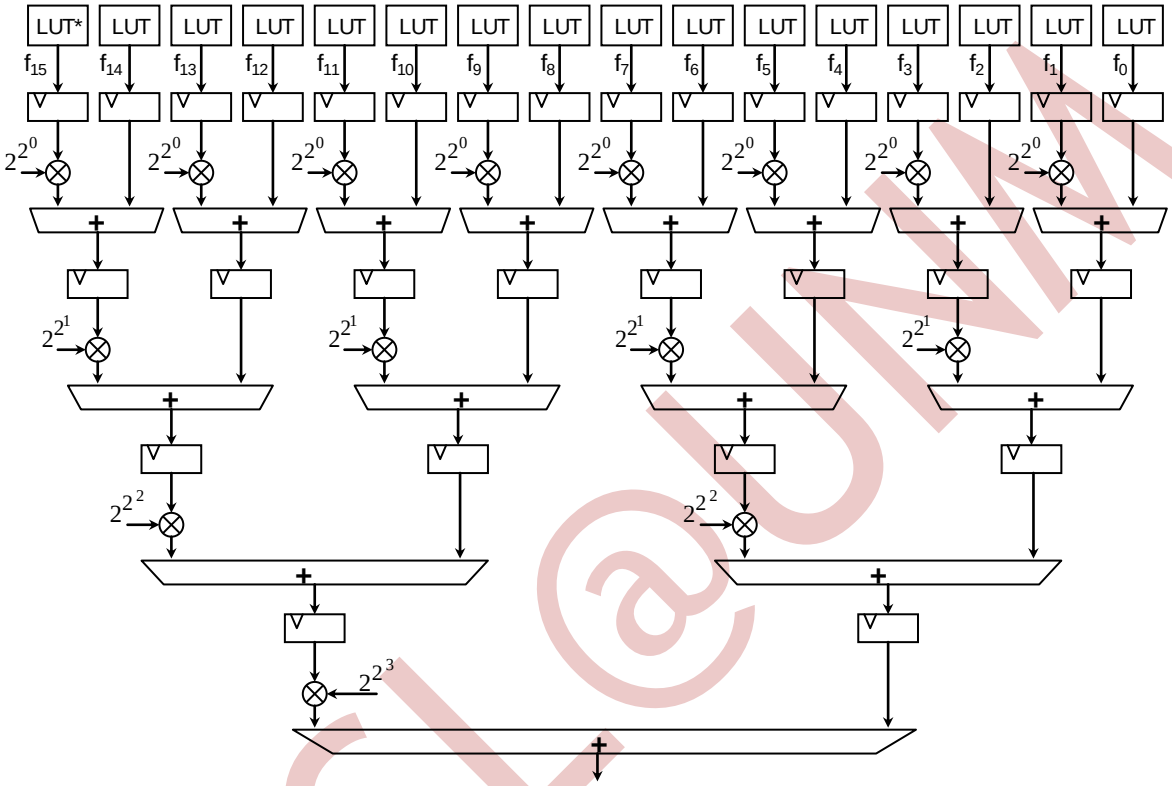

Figure 13. Filterblock realization: full tree adder with shifting spread over the adder levels

Figure 13 shows how we embedded the multiplication by 2^b along the $\lceil log_2 16 \rceil = 4$ adder levels. This arrangement arises from the following derivation:

$$\sum_{b=0}^{15} f_b 2^b = f_{15}2^{15} + f_{14}2^{14} + f_{13}2^{13} + f_{12}2^{12} + f_{11}2^{11} + f_{10}2^{10} + f_9 2^9 + f_8 2^8 + f_7 2^7 + f_6 2^6 + f_5 2^5 + f_4 2^4 + f_3 2^3 + f_2 2^2 + f_1 2^1 + f_0 2^0$$

$$= 2^{14}\left(f_{15}2^1 + f_{14}\right) + 2^{12}\left(f_{13}2^1 + f_{12}\right) + 2^{10}\left(f_{11}2^1 + f_{10}\right) + 2^8\left(f_9 2^1 + f_8\right) + 2^6\left(f_7 2^1 + f_6\right) + 2^4\left(f_5 2^1 + f_4\right) + 2^2\left(f_3 2^1 + f_2\right) + 2^0\left(f_1 2^1 + f_0\right)$$

$$= 2^{12}\left(2^2\left(f_{15}2^1 + f_{14}\right) + \left(f_{13}2^1 + f_{12}\right)\right) + 2^8\left(2^2\left(f_{11}2^1 + f_{10}\right) + \left(f_9 2^1 + f_8\right)\right) + 2^4\left(2^2\left(f_7 2^1 + f_6\right) + \left(f_5 2^1 + f_4\right)\right) + 2^0\left(2^2\left(f_3 2^1 + f_2\right) + \left(f_1 2^1 + f_0\right)\right)$$

$$= 2^8\left[2^4\left(2^2\left(f_{15}2^1 + f_{14}\right) + \left(f_{13}2^1 + f_{12}\right)\right) + \left(2^2\left(f_{11}2^1 + f_{10}\right) + \left(f_9 2^1 + f_8\right)\right)\right] + 2^0\left[2^4\left(2^2\left(f_7 2^1 + f_6\right) + \left(f_5 2^1 + f_4\right)\right) + \left(2^2\left(f_3 2^1 + f_2\right) + \left(f_1 2^1 + f_0\right)\right)\right]$$

$$= 2^0\left\{2^8\left[2^4\left(2^2\left(f_{15}2^1 + f_{14}\right) + \left(f_{13}2^1 + f_{12}\right)\right) + \left(2^2\left(f_{11}2^1 + f_{10}\right) + \left(f_9 2^1 + f_8\right)\right)\right] + \left[2^4\left(2^2\left(f_7 2^1 + f_6\right) + \left(f_5 2^1 + f_4\right)\right) + \left(2^2\left(f_3 2^1 + f_2\right) + \left(f_1 2^1 + f_0\right)\right)\right]\right\}$$

This arrangement requires the right summand not to be shifted.

We see that: At the first level, we multiply by 2^{2^0} (or shift by 2^0)

At the second level, we multiply by 2^{2^1} (or shift by 2^1)

At the third level, we multiply by 2^{2^2} (or shift by 2^2)

→ At the 'ith' level, we multiply by 2^{2^i} (or shift by 2^i) (proof by induction).

At the last level (level $log_2(sizeI)$), the shifting is by $2^{log_2(sizeI)-1}$.

- $f_{sizeI-1}$ gets the maximum shifting (by $sizeI-1$). To demonstrate that this can be done in $log_2(sizeI)$ adder tree levels, let's call 'p' the maximum shifting caused.

  → $2^p = 2^{2^0} 2^{2^1} 2^{2^2} \cdots 2^{2^{log_2(sizeI)-1}}$ (this arises from the fact that $f_{sizeI-1}$ gets shifted at every level).

  → $p = 2^0 + 2^1 + \cdots + 2^{log_2(sizeI)-1} = \sum_{i=0}^{log_2(sizeI)-1} 2^i = \frac{2^{log_2(sizeI)}-1}{2-1} = sizeI-1$

  Then: $p = sizeI-1$, which is in fact the maximum shifting possible with $sizeI$ bits.

  Thus, it has been demonstrated that to spread the multiplication by $2^{sizeI-1}$ as shown in Figure 13, we need exactly $log_2(sizeI)$ adder levels.

- If $sizeI$ is not a multiple of 2:

  We know that we need $\lceil log_2(sizeI) \rceil$ levels. We can make use of the rearrangement shown in Figure 13: if we assume that the tree is not full, we note that each f_b will still be multiplied by 2^b.

  The rearrangement proceeds as this:
  - We move from right to left: we only place shifters in the left summand (as in the full tree case)
  - In case we find a solitary term at the leftmost position, we do not shift it, and rather we pass that as the result to the next level.

  So, the idea works for every $sizeI$ (> 1) (demonstration is not provided here).

Finally recall that the number of adder levels is $\lceil log_2(sizeI) \rceil$. In Figure ??, we included pipelining registers, and the number of registers levels is the same as the number of register levels. Then:

$\boxed{\#of\ register levels in\ Filter Block = \lceil log_2(sizeI) \rceil}$ (more useful since the levels are better understood with registers).

Figure 14 shows an actual filter block pipelined realization for a specific case. The multiplication by 2^b (a shift to the left) is optimized along with the final summation into a tree-like structure. The filter block consists of an L-input array, an adder tree, shifters, and registers.

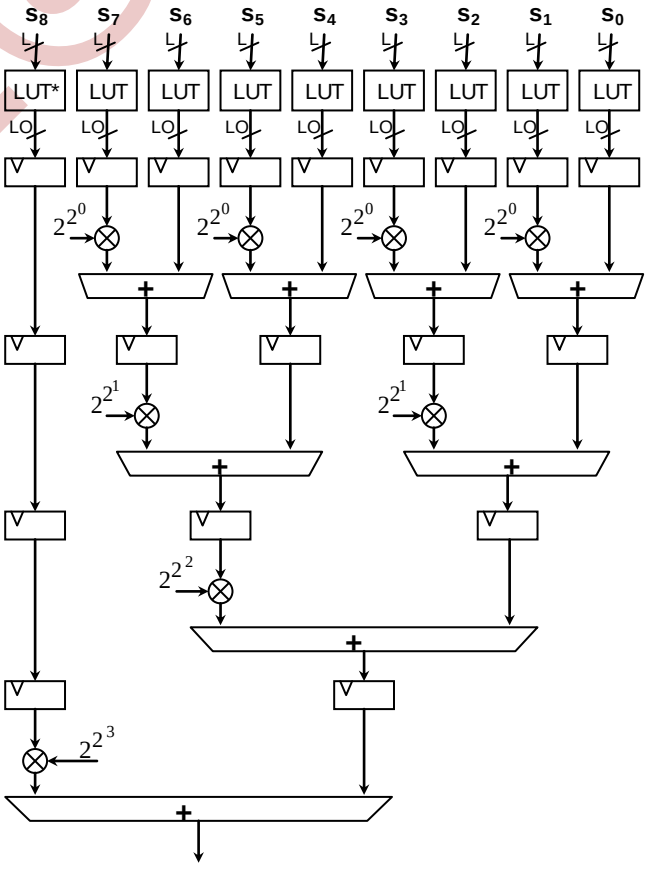

Figure 14. Filter block realization. SYMMETRY=YES. B = 8

IV. L-INPUT LUT IMPLEMENTATION

We have M/L filter blocks. Each of them contains $sizeI$ L-input LUTs. Figure 15 shows an LUT, 'i' indicates the filter block to which the LUT belongs: $0 \le i \le {}^{M}\!/_{L} - 1$

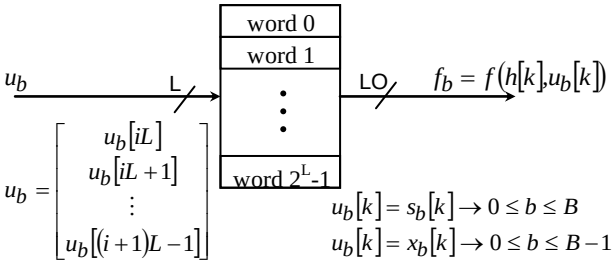

Figure 15.LUT with L inputs and LO outputs

An L-input LUT performs the following computation:
Symmetric filter:

$$f_b = f(h[k], s_b[k]) = \begin{cases} \sum_{k=iL}^{(i+1)L-1} h[k]s_b[k], & 0 \le b \le B-1 \\ -\sum_{k=iL}^{(i+1)L-1} h[k]s_b[k], & b = B \end{cases}$$

Non-symmetric filter:

$$f_b = f(h[k], x_b[k]) = \begin{cases} \sum_{k=iL}^{(i+1)L-1} h[k]x_b[k], & 0 \le b \le B-2 \\ -\sum_{k=iL}^{(i+1)L-1} h[k]x_b[k], & b = B-1 \end{cases}$$

LUT CONSTRUCTION

The LUT values are defined by the values of $u_b[k], iL \le k \le (i+1)L-1$. To see how the LUT values are created, let's use 2 examples: one with L = 4 and the other with L = 5.

Without loss of generality, let's use SYMMETRY = YES and $i = 0$ (1st filter block).
L = 4: Table 6 shows the LUT contents for $0 \le b \le B-1$. If $b = B$, we invert the sign of those values.

TABLE 6. LUT VALUES FOR L = 4, $0 \le b \le B-1$

$s_b[3]$	$s_b[2]$	$s_b[1]$	$s_b[0]$	$\sum_{k=0}^{3} h[k]s_b[k]$
0	0	0	0	0
0	0	0	1	$h[0]$
0	0	1	0	$h[1]$
0	0	1	1	$h[1]+h[0]$
0	1	0	0	$h[2]$
0	1	0	1	$h[2]+h[0]$
0	1	1	0	$h[2]+h[1]$
0	1	1	1	$h[2]+h[1]+h[0]$
1	0	0	0	$h[3]$
1	0	0	1	$h[3]+h[0]$
1	0	1	0	$h[3]+h[1]$
1	0	1	1	$h[3]+h[1]+h[0]$
1	1	0	0	$h[3]+h[2]$
1	1	0	1	$h[3]+h[2]+h[0]$
1	1	1	0	$h[3]+h[2]+h[1]$
1	1	1	1	$h[3]+h[2]+h[1]+h[0]$

L = 5: Table 7 shows the LUT contents for $0 \le b \le B-1$. If $b = B$, we invert the sign of those values.

TABLE 7. LUT VALUES FOR L = 5, $0 \le b \le B-1$

$s_b[4]$	$s_b[3]$	$s_b[2]$	$s_b[1]$	$s_b[0]$	$\sum_{k=0}^{4} h[k]s_b[k]$
0	0	0	0	0	0
0	0	0	0	1	$h[0]$
0	0	0	1	0	$h[1]$
0	0	0	1	1	$h[1]+h[0]$
0	0	1	0	0	$h[2]$
0	0	1	0	1	$h[2]+h[0]$
0	0	1	1	0	$h[2]+h[1]$
0	0	1	1	1	$h[2]+h[1]+h[0]$
0	1	0	0	0	$h[3]$
0	1	0	0	1	$h[3]+h[0]$
0	1	0	1	0	$h[3]+h[1]$
0	1	0	1	1	$h[3]+h[1]+h[0]$
0	1	1	0	0	$h[3]+h[2]$
0	1	1	0	1	$h[3]+h[2]+h[0]$
0	1	1	1	0	$h[3]+h[2]+h[1]$
0	1	1	1	1	$h[3]+h[2]+h[1]+h[0]$
1	0	0	0	0	$h[4]$
1	0	0	0	1	$h[4]+h[0]$
1	0	0	1	0	$h[4]+h[1]$
1	0	0	1	1	$h[4]+h[1]+h[0]$
1	0	1	0	0	$h[4]+h[2]$
1	0	1	0	1	$h[4]+h[2]+h[0]$
1	0	1	1	0	$h[4]+h[2]+h[1]$
1	0	1	1	1	$h[4]+h[2]+h[1]+h[0]$
1	1	0	0	0	$h[4]+h[3]$
1	1	0	0	1	$h[4]+h[3]+h[0]$
1	1	0	1	0	$h[4]+h[3]+h[1]$
1	1	0	1	1	$h[4]+h[3]+h[1]+h[0]$
1	1	1	0	0	$h[4]+h[3]+h[2]$
1	1	1	0	1	$h[4]+h[3]+h[2]+h[0]$
1	1	1	1	0	$h[4]+h[3]+h[2]+h[1]$
1	1	1	1	1	$h[4]+h[3]+h[2]+h[1]+h[0]$

DERIVATION OF THE NUMBER OF OUTPUT BITS FOR THE L-INPUT LUT

Without loss of generality, we will use $i = 0$.

Case 'b' is not the MSB:

The worst case occurs when $u_b[k] = 1, 0 \le k \le L-1$ → $f_b = \sum_{k=0}^{L-1} h[k]\,u_b[k] = \sum_{k=0}^{L-1} h[k]$

- L is a power of 2: We can derivate the number of output bits for the LUTs with a full tree adder that computes the summation (see Fig. 16):

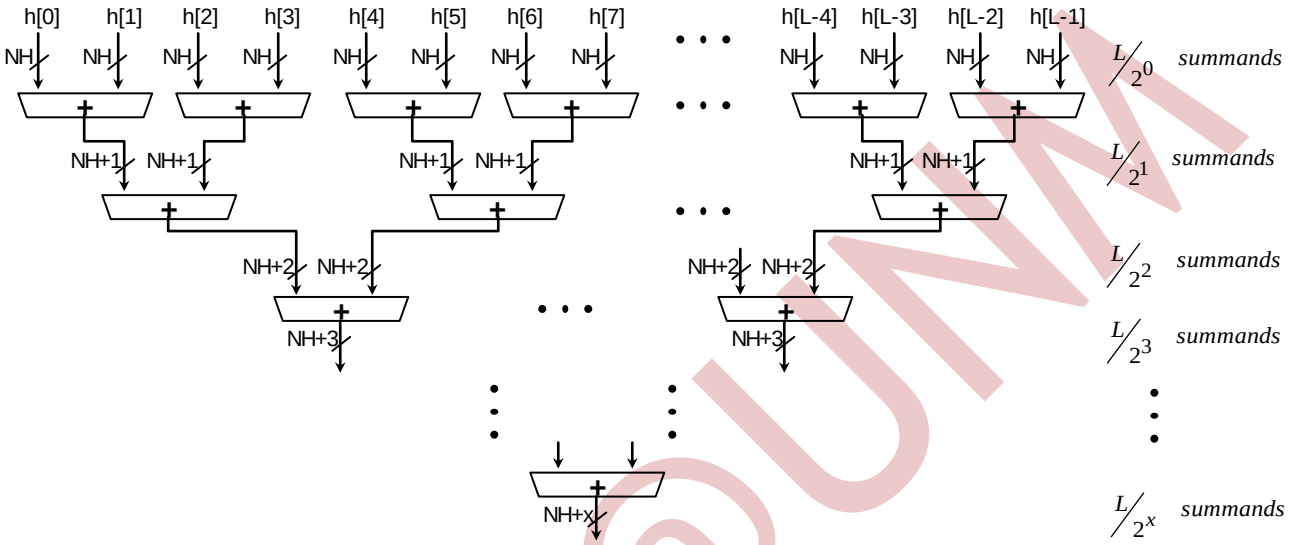

Figure 16. Adder tree: determining bit increase and number of adder levels

$LUT\ output bits = LO = NH + x$, $\#\ of adder\ levels = x$

Each level adds 1 bit to the word-length, up to 'x' bits. We get 'x' by noting that there is only one summand at the end:

→ $L/2^x = 1 \rightarrow x = log_2 L$

- If L is not a power of 2: The tree adder is not full. Let's define 'Lp' to be the largest power of 2 whose value is lower than L → $Lp < L < 2Lp$ ➔ $log_2 Lp < log_2 L < log_2 2Lp \rightarrow log_2 Lp < log_2 L < log_2 Lp + 1$.

  We can conclude that $\lceil log_2 L \rceil = log_2 2Lp$ (*)

  Also: Lp summands require $log_2 Lp$ adder levels.

  $2Lp$ summands require $log_2 2Lp = log_2 Lp + 1$ adder levels, i.e. just 1 extra level.

  L summands require 'x' adder levels.

  'x' has to be definitely greater than $log_2 Lp$, but as an integer, its next possible value ($log_2 Lp + 1$)is the number of adder levels $2Lp$ summands require, i.e. $x = log_2 2Lp$ (**)

  From (*) and (**), we have that: $x = \lceil log_2 L \rceil$.

Finally, for any L: Number of adder levels (or extra bits): $x = \lceil log_2 L \rceil$

Number of LUT output bits: $LO = NH + \lceil log_2 L \rceil$.

Case 'b' is the MSB:

In this case, we need to change the sign of the final summation.

We have: $f_B = -\sum_{k=0}^{L-1} h[k]$ for symmetric filters, and $f_{B-1} = -\sum_{k=0}^{L-1} h[k]$ for non-symmetric filters.

Recall that: $-2^{NH-1} \le h[k] \le 2^{NH-1} - 1$, h[k]: 2's complement representation with 'NH' bits.

As we know, $SUM\_LUT = \sum_{k=0}^{L-1} h[k]$ requires $LO = NH + \lceil log_2 L \rceil$ bits. To obtain the number of bits $-\sum_{k=0}^{L-1} h[k]$ needs, we analyze the two extreme cases:

- $h[k] = 2^{NH-1} - 1, 0 \le k \le L-1$ (largest positive number): Then, $SUM\_LUT = L \times 2^{NH-1} > 0$. In 2's complement, the negative of a positive number does not need an extra bit. Therefore, $-SUM\_LUT$ requires LO bits.
- $h[k] = -2^{NH-1}, 0 \le k \le L-1$ (largest negative number): Then, $SUM\_LUT = -L \times 2^{NH-1} < 0$.

  By definition: $-2^{NH+\lceil log_2 L \rceil - 1} \le SUM\_LUT \le 2^{NH+\lceil log_2 L \rceil - 1} - 1$, since it uses LO bits.

  We have that: $SUM\_LUT = -2^{log_2 L} \times 2^{NH-1} = -2^{NH-1+log_2 L}$ → $-SUM\_LUT = 2^{NH-1+log_2 L}$

  Two cases arise here:
  - L is not a multiple of 2: Then $-SUM\_LUT = 2^{NH-1+log_2 L} < 2^{NH+\lceil log_2 L \rceil - 1}$ (alternatively we can say that SUM_LUT is NOT the largest negative number). This means that $-SUM\_LUT$ is within the bounds of SUM_LUT, and as such requires LO bits.
  - L is a multiple of 2: Then $-SUM\_LUT = 2^{NH-1+log_2 L} = 2^{NH+\lceil log_2 L \rceil - 1}$ (alternatively we can say that SUM_LUT is the largest negative number). This means that $-SUM\_LUT$ is out of the bounds of SUM_LUT (by one LSB), and as such requires 'LO+1' bits.

  This last case is rare since it requires: $u_b[k] = 1, h[k] = -2^{NH-1}, 0 \le k \le L-1, b = MSB$, and L to be multiple of 2. We avoid having to increase one bit just for this very special case by subtracting 1 LSB to $-SUM\_LUT$ so that $-SUM\_LUT = 2^{NH+\lceil log_2 L \rceil - 1} - 1$, allowing it to use 'LO' bits. The LUT* will have an incorrect value (by 1 LSB) if this special case arises. The quantization noise introduced by this modification is unknown, but we expect it not to be significant. Also, in the next section, we will discover that storing the LUT values rather than the coefficient themselves leads to a system slightly less sensitive to quantization noise.

In conclusion, the number of output bits is defined by: $\boxed{LO = NH + \lceil log_2 L \rceil}$

L-INPUT LUT DECOMPOSITION INTO L-TO-1 LUTS

Figure 17 shows the structure of a L-input LUT. The word size (output bits) of each L-input LUT was computed to be $LO = NH + \lceil log_2(L) \rceil$. We also show its decomposition into LO L-to-1 LUTs, useful for efficient FPGA implementation. Xilinx® FPGA devices contain L-to-1 LUT primitives with L = 4 (Spartan 3, Virtex-II Pro, Virtex-4) and L = 6 (Virtex-5). Moreover, we can also obtain LUT implementations for L = 4, 5, 6, 7, 8.

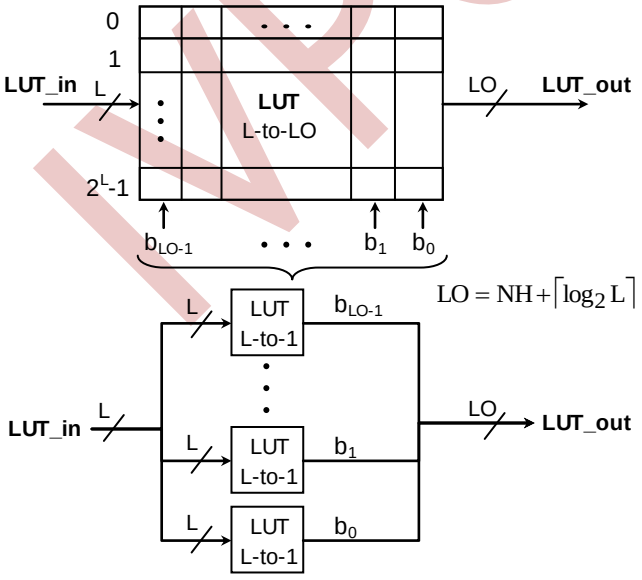

Figure 17. Decomposition of a L-to-LO LUT into LO L-to-1 LUTs

CREATING THE TEXT FILE THAT HOLDS THE LUT VALUES

Figure 18 depicts the structure of the text file used to store the LUT values. For each filterblock, we store 2 sets of values:

$$f_b = f\left(h[k], u_b[k]\right) = \sum_{k=iL}^{(i+1)L-1} h[k]u_b[k], \quad 0 \le b \le MSB-1$$

$$f_{MSB} = f\left(h[k], u_{MSB}[k]\right) = -\sum_{k=iL}^{(i+1)L-1} h[k]u_{MSB}[k], \quad b = MSB$$

$$0 \le i \le M/L-1$$

Also, between sets, we include a blank line just for clarity purposes. Table 8 indicates the line position in the file (starts at 0) given the filter block number and whether 'b' is the MSB.

TABLE 8. LINE POSITION FOR FILTER BLOCK NUMBER

Filter block	b	Line position
0	no MSB	0
	MSB	2^L+1
1	no MSB	$2\left(2^L+1\right)$
	MSB	$2\left(2^L+1\right)+2^L+1$
2	no MSB	$4\left(2^L+1\right)$
	MSB	$4\left(2^L+1\right)+2^L+1$
. . .	. . .	. . .
i	no MSB	$2\times i\left(2^L+1\right)$
	MSB	$2\times i\left(2^L+1\right)+2^L+1$
. . .	. . .	. . .
$M/L-1$	no MSB	$2\times(M/L-1)\left(2^L+1\right)$
	MSB	$2\times(M/L-1)\left(2^L+1\right)+2^L+1$

As a result, for a given filter block 'i'', we have that the position is defined by:

$$Line\ position = 2\times i\times\left(2^L+1\right)+(b==MSB)\times\left(2^L+1\right),\ 0 \le i \le M/L-1$$

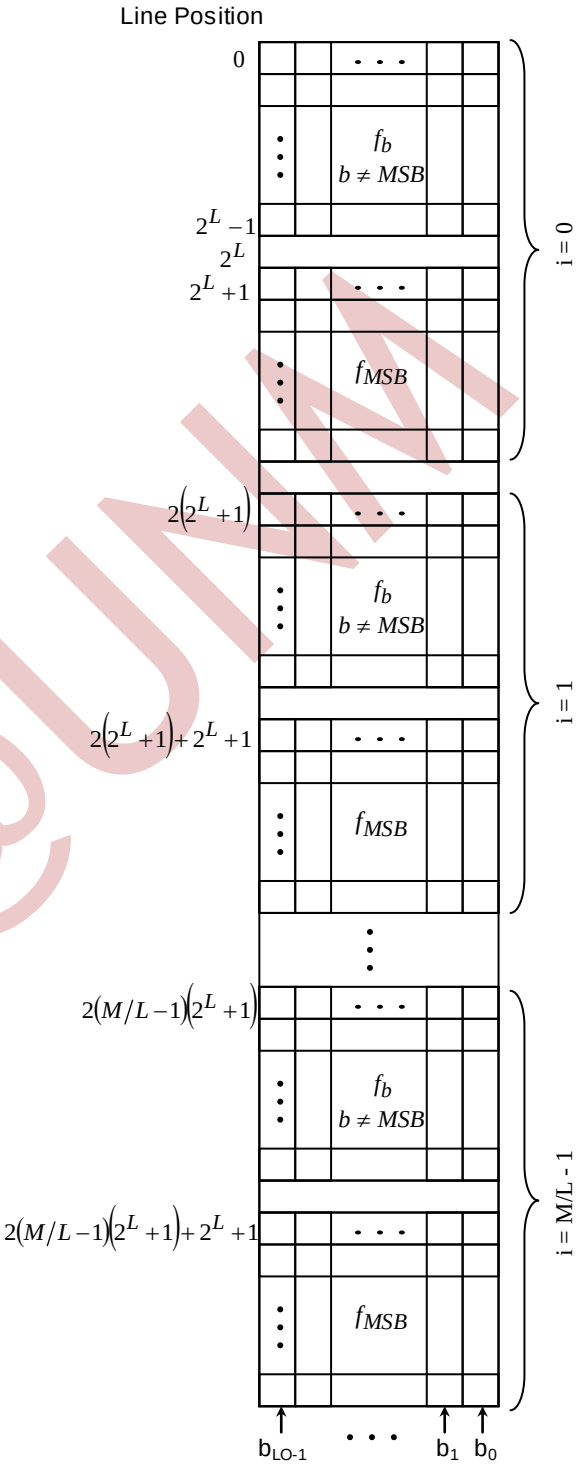

Figure 18. Text file construction

- Important: Even though the coefficients are restricted to 'NH' bits, we note that the FIR filter core does not use the coefficients directly, but rather we use the LUT values (see Tables 6 and 7). We have determined the LUT values bit-width to be $NH+log_2 L$. Then, we do not need to quantize the coefficients, but rather the LUT values, leading to slightly more accurate results. Therefore, this FIR DA Implementation is slightly less sensitive to quantization noise than a normal implementation, with quantized coefficients.

V. FINAL ADDER TREE

Fig. 19 depicts the pipelined architecture of the extra adder tree that adds up all the Filter block outputs. A final output register is also shown. The number of summands is equal to M/L. Then, using the result from the previous section, we need $\lceil log_2(M/L) \rceil$ adder levels. We introduce pipeline registers, and the number of register levels is the same as the number of adder levels:

$$\#of\ register levels in\ Final\ Adder Tree = \lceil log_2(M/L) \rceil$$

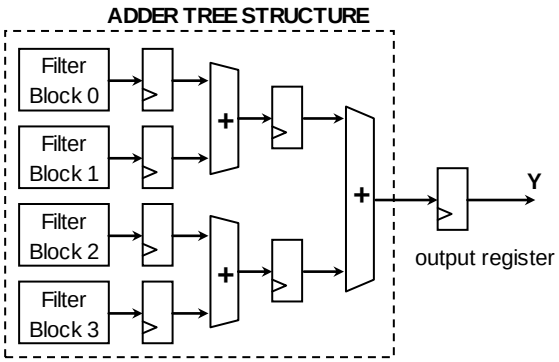

Figure 19. Final adder tree. M/L=4

VI. INPUT-OUTPUT DELAY

The FIR filter architecture has an input-output delay of $REG\_LEVELS = \lceil log_2(sizeI) \rceil + \lceil log_2(M/L) \rceil + 2$ cycles, i.e. REG_LEVELS is the number of registers between input and output (see Fig. 20).

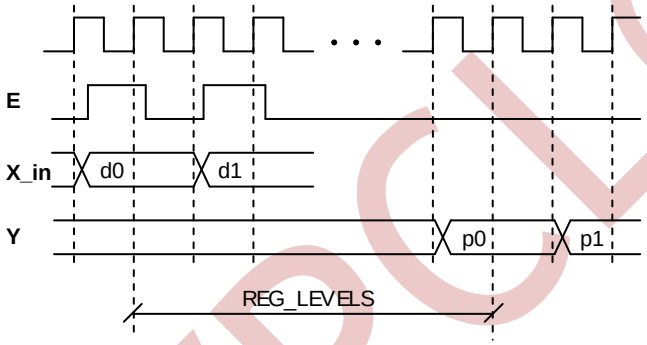

Figure 20. Latency measured from the moment 'd0' is input until its correspondent output 'p0' is available

VII. FIXED-POINT FORMAT CONSIDERATIONS

INPUT/OUTPUT FORMAT

- Assumption: The coefficient and input values are constrained to $[-1,1\rangle$, so that the fixed-point representation only includes one integer bit: $Format(h[k]) = [NH \quad NH-1]$, $Format(x[k]) = [B \quad B-1]$
- Figure 21 illustrates the derivation of the output format given the inputs and coefficients input format.
- It is important to remember that the Distributed Arithmetic formulas are intended for integer values (we could modify it for fractional values, but it is not worth the effort). As a result, the FIR Filter core treats every word as integer. To transform the integer values into fractional ones, we just need to correctly locate the fractional point. This amounts to performing a preprocessing of the inputs and post-processing of the output (see Figure 22).
- The preprocessing and post-processing are virtual steps since nothing needs to be done in hardware. We only need to consider where to place the output fractional point when retrieving the outputs, i.e. we must be aware of the output fixed-point format before attempting to perform further operations.

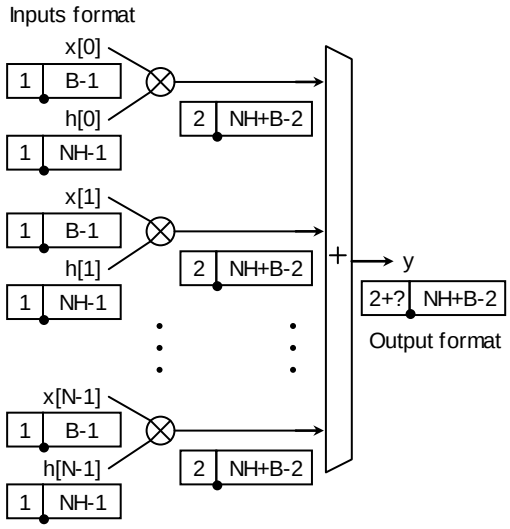

Figure 21. Output format derivation

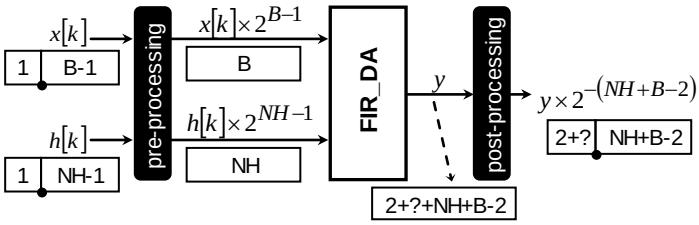

Figure 22. Pre-processing and post-processing modules for correct output results

DERIVATION OF THE MAXIMUM WORD-LENGTH FOR FILTER OUTPUT

The filter is just the sum of N products, so we just need to see how many bits that operation needs based on the word-length of the inputs and coefficients. As explained above, the FIR Filter core considers its operands as integers in 2's complement representation:

→ $-2^{B-1} \le x[k] \le 2^{B-1}-1 \quad (B\ bits)$

$-2^{NH-1} \le h[k] \le 2^{NH-1}-1 \quad (NH\ bits)$

Then we perform the following operation: $y = \sum_{k=0}^{N-1} x[k] \times h[k]$ (summation of N products)

We see that: Largest positive value: $\left(-2^{NH-1}\right)\left(-2^{B-1}\right)N = 2^{NH+B-2}N$

Largest negative value: $\begin{cases} \left(-2^{NH-1}\right)\left(2^{B-1}-1\right)N = -2^{NH+B-2}N + 2^{NH-1}N, if\ NH \le B \\ \left(-2^{B-1}\right)\left(2^{NH-1}-1\right)N = -2^{NH+B-2}N + 2^{B-1}N, if\ B < NH \end{cases}$

→ $-2^{NH+B-2}N + min\left(2^{NH-1}N, 2^{B-1}N\right) \le y \le 2^{NH+B-2}N \equiv -2^{NH+B-2}N < y \le 2^{NH+B-2}N$

i) <u>N is a power of 2</u>: $N = 2^n$:

Then: $-2^{NH+B-2+n} < y \le 2^{NH+B-2+n}$

Recall: $If\ \#bits\ of\ v = x \rightarrow -2^{x-1} \le v \le 2^{x-1}-1$ (γ)

If $-2^{NH+B-2+n} < y \le 2^{NH+B-2+n}-1$, then 'y' would require $NH+B-2+n+1$ 'NH+B-2+n+1'bits. However, we miss it by one LSB, and as a result, we need $NH+B-2+n+2 = NH+B+log_2 N$ bits.

ii) <u>N is not a power of 2</u>:

Let's define Np to be the immediate power of 2 after N → $Np/2 < N < Np$ → $log_2(Np/2) < log_2 N < log_2 Np$

Then, we conclude that $\lceil log_2 N \rceil = log_2 Np$, and $Np = 2^{\lceil log_2 N \rceil}$. Also: $2^{\lceil log_2 N \rceil - 1} < N < 2^{\lceil log_2 N \rceil}$

Now: $-2^{NH+B-2+\lceil log_2 N \rceil} < -2^{NH+B-2}N < y \le 2^{NH+B-2}N < 2^{NH+B-2+\lceil log_2 N \rceil}$ (we only use $N < 2^{\lceil log_2 N \rceil}$

From (γ), we see that we need $NH+B-2+\lceil log_2 N \rceil + 1 = NH+B+\lceil log_2 N \rceil - 1$ bits.

We see that if N is a power of 2, we have that: $log_2 N = \lceil log_2(N+1) \rceil - 1$

And if N is not a power of 2, we have that: $\lceil log_2 N \rceil = \lceil log_2(N+1) \rceil$

From (i), and (ii), we conclude that: $\boxed{L\_FIR = NH+B+\lceil log_2(N+1) \rceil - 1}$ (maximum # of bits for the filter).

DERIVATION OF THE MAXIMUM WORD-LENGTH FOR FILTER BLOCK OUTPUT

The filter block is just the sum of L products, so we just need to see how many bits that operation needs based on the word-length of the input sequence and coefficients.

→ $-2^{sizeI-1} \le u[k] \le 2^{sizeI-1}-1 \quad (sizeI\ bits)$

$-2^{NH-1} \le h[k] \le 2^{NH-1}-1 \quad (NH\ bits)$

This case is identical to the case of the previous subsection. The number of products is L instead of N, and the size of the input is $sizeI$ instead of B.

Then: $\boxed{L\_fbk = NH + sizeI + \lceil log_2(L+1) \rceil - 1}$ (maximum # of bits a filter block needs)

WORD-LENGTH OF INTERMEDIATE OPERATIONS IN THE FILTERBLOCK

Figure 23 shows a filter block implementation with the signals nomenclature. Even though we know that the final result is no larger than L_fbk, we do not know the size of internal datapath at every register level. We come up with the signals' word-length given the register level.

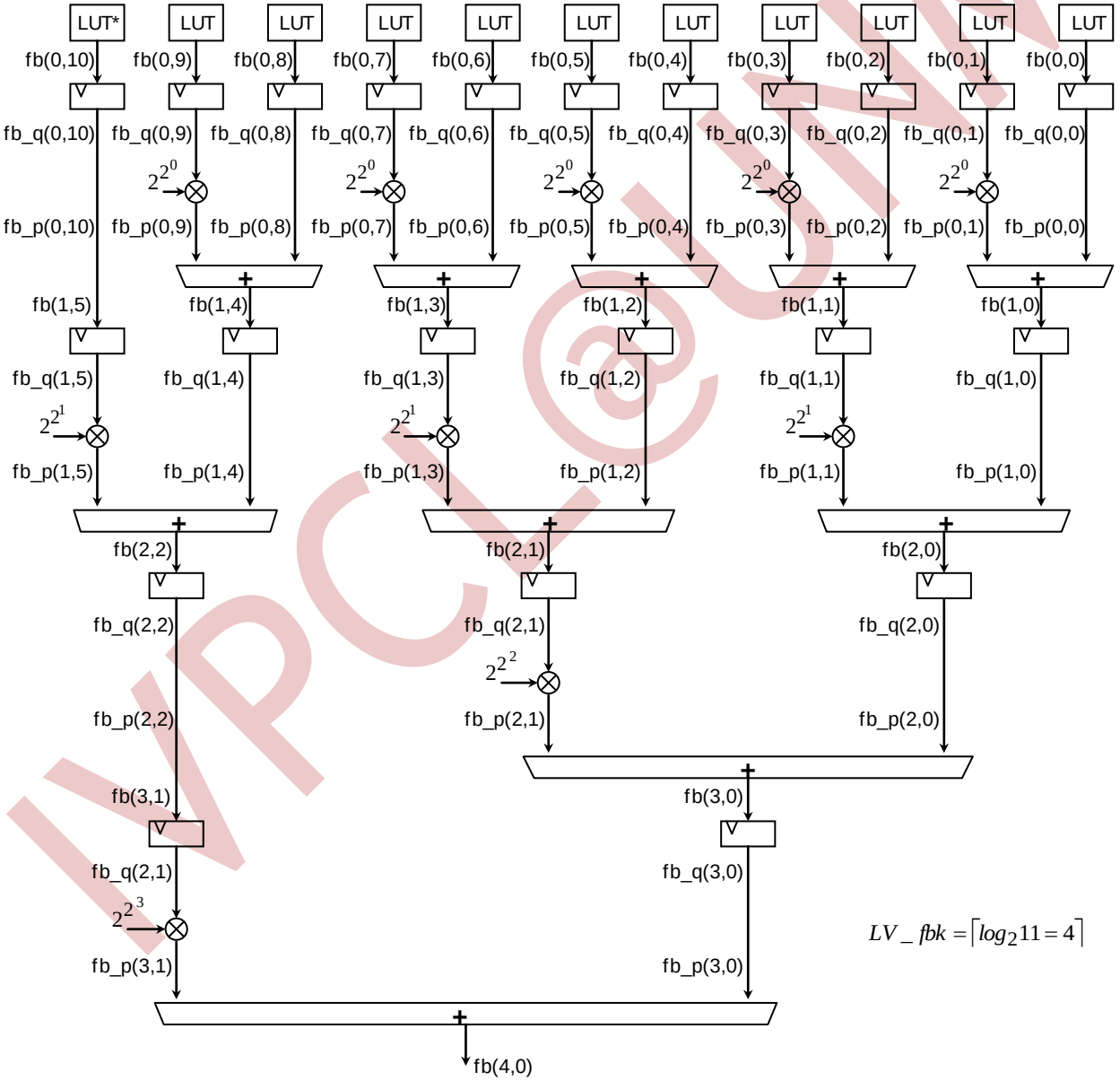

Figure 23. Filter block implementation with the respective signals at every level. sizeI = 11

Each signal has two indices (starting at zero). The first index indicates the register/adder level the signal is at, and the second index indicates the horizontal position (from the right).

$LV\_fbk = \lceil log_2(sizeI) \rceil$ (number of register (or adder) levels)

$Length\{fb(0,:)\} = NH + \lceil log_2 L \rceil = LO$

$fb(i,:)$: $0 \le i \le LV\_fbk$ ($i = LV\_fbk$ is only allowed for this signal, because we made an exception at $i = 0$, where $fb(0,:)$ represents the LUTs outputs rather than the adders outputs).

$fb\_q(i,:)$: $0 \le i \le LV\_fbk - 1$ (registers' outputs)

$fb\_p(i,:)$: $0 \le i \le LV\_fbk - 1$ (adders' inputs)

Figure 24 shows us the increments at every adder output level: $fb(i,:)$, $1 \le i \le LV\_fbk$. Recall that a left shift to a negative value may result in the largest negative value at the new wordlength. As a result, the summation of the shifted quantity with the non-shifted quantity requires inevitably one more bit than the shifted quantity.

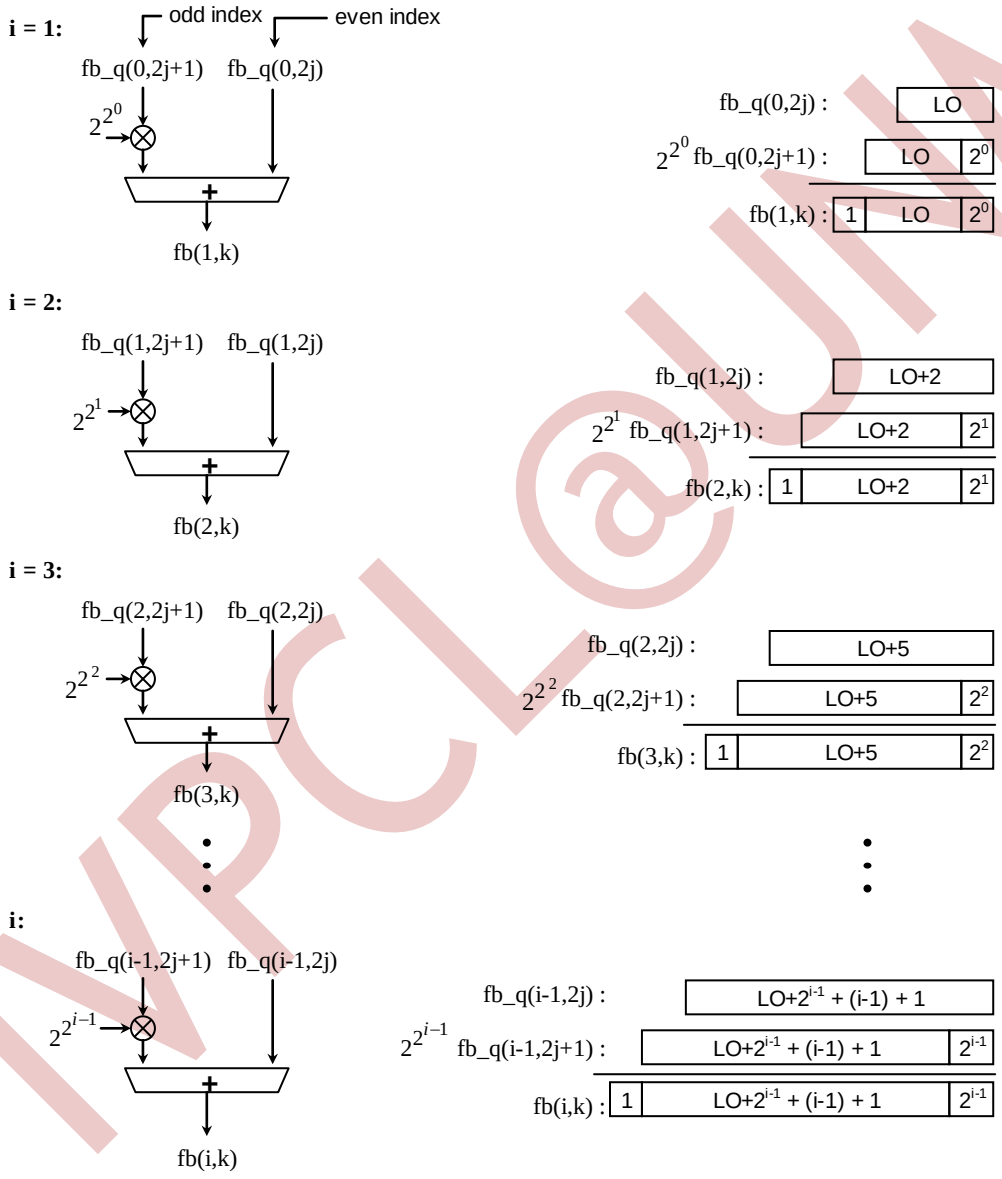

Figure 24. Bit increments per level

TABLE 9. INCREMENTS AT LEVEL 'i' FROM THE PREVIOUS LEVEL '$i-1$'

Level	1	2	3	…	i
Bit increments at every level for $fb(i,:)$, $1 \le i \le LV\_fbk$	2^0+1	2^1+1	2^2+1	$\cdots$	$2^{i-1}+1$

Table 9 shows how many bits we add at level i from the previous level $i-1$. Our goal is to know the total bit increment at every level i from the initial level, i.e. if $Len\{fb(i,:)\} = LO + g(i)$, we want to know $g(i)$.
This can be determined by adding up all the previous increments:

→ $g(i) = \sum_{k=0}^{i-1}\left(2^k + 1\right) = i + \sum_{k=0}^{i-1} 2^k = i + 2^i - 1 = 2^i + i - 1$ ➔ $\boxed{Len\{fb(i,:)\} = NH + \lceil log_2 L \rceil + 2^i + i - 1}$

Figure 25 shows how the $fb\_p(i,:):\ 0 \le i \le LV\_fbk-1$ signals should be implemented. They have the same length as the adder output because the LPM adders used in this implementation require all the operands to have the same bit-width. Also, all signals at a level have the same bit-width in order to ease implementation.

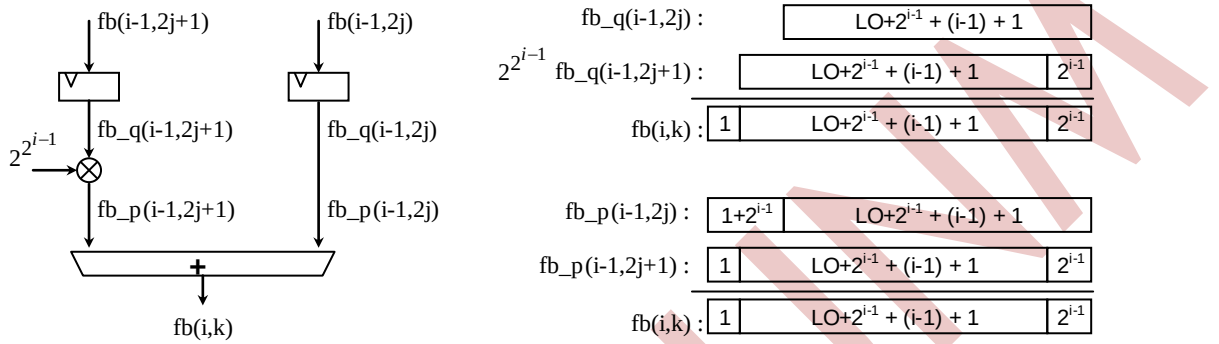

Figure 25. Implementation of 'fb_p' signals

- Recall that the maximum number of bits needed is L_fbk. The smaller summations are also bounded by L_fbk (the summation includes the shifters).
- When designing the adder levels, it is possible that the internal signals get bigger than L_fbk. In order to optimize this process to make sure that no internal signal uses more than L_fbk bits, we use the following algorithm (use Fig. 25)

If $Len\{fb(i-1,:)\} < L\_fbk$ *then*
  If $Len\{fb(i,:)\} \le L\_fbk$ *then*
    No change for: $fb\_p(i-1,:)$
  else
    TruncateMSBs of: $fb\_p(i-1,:)$ *so that it has* L_fbk *bits*
  end
else
  TruncateMSBs of: $fb\_p(i-1,:)$ *so that it has* L_fbk *bits*
end

WORD-LENGTH OF INTERMEDIATE OPERATIONS IN THE FINAL ADDER TREE

This case is easier than the previous one since each level adds exactly one bit to the final result. The construction of the adder tree is similar too, but here we do not have the shifters. Even though we know that the final result is no larger than L_FIR, we do not know the size of internal datapath at every register level. Keep in mind that the number of bits of the FIR block outputs is L_fbk.

$LV = \lceil log_2(M/L) \rceil$ (number of register (or adder) levels)
$Length\{yf(0,:)\} = L\_fbk$
$yf(i,:):\ 0 \le i \le LV$ ($i = LV$ is only allowed for this signal, because we made an exception at $i = 0$, where $yf(0,:)$ represents the Filter block outputs rather than the adders outputs).
$yf\_q(i,:):\ 0 \le i \le LV-1$ (registers' outputs)
$yf\_p(i,:):\ 0 \le i \le LV-1$ (adders' inputs)

Figure 26 shows the final adder tree implementation along with the signals nomenclature at every level.

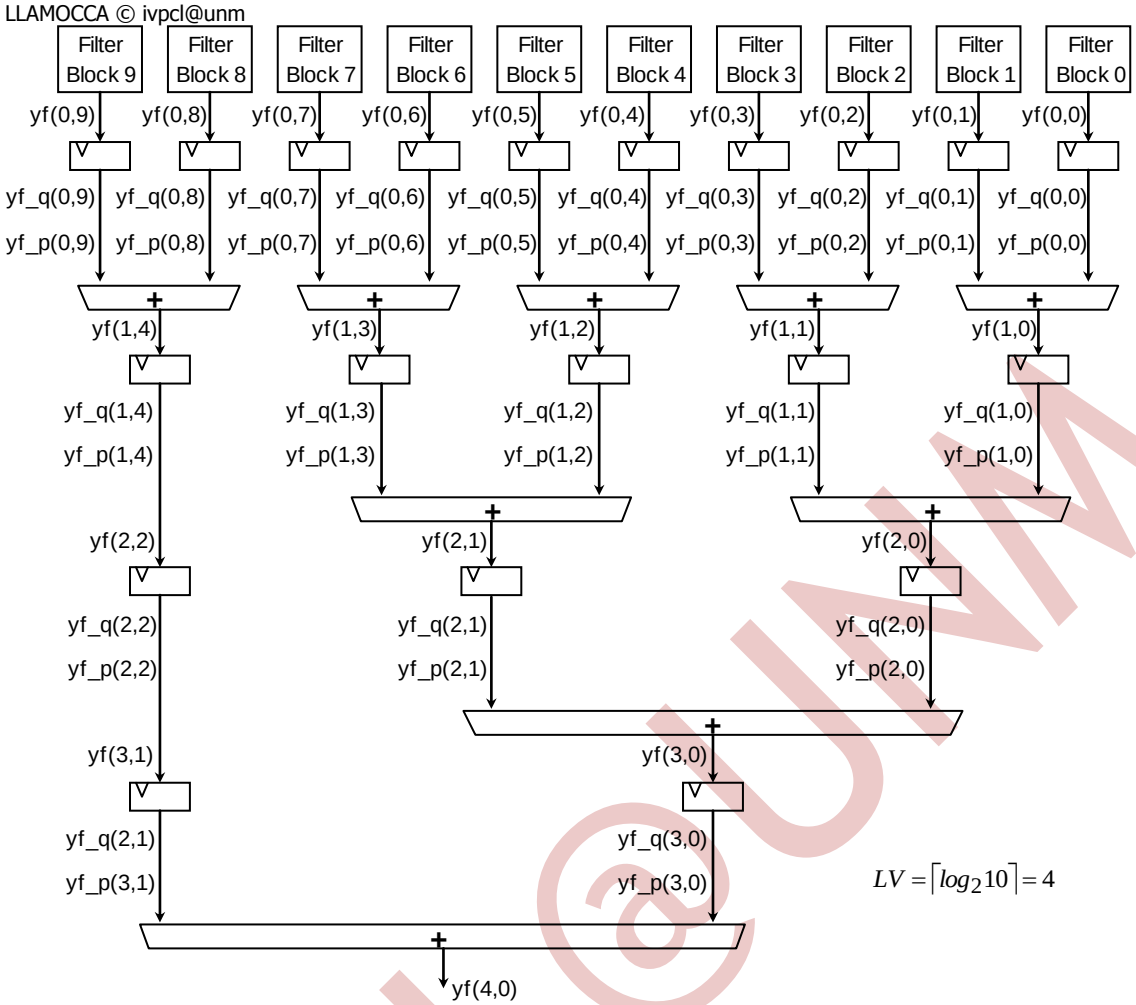

Figure 26. Final adder implementation with the respective signals at every level. M/L = 10

Figure 27 shows how the signals should be implemented. The fact that 'yf_p' has the same length as the adder output is because the LPM adders used in this implementation require all the operands to have the same bit-width. Also, all signals at a level have the same bit-width in order to ease implementation

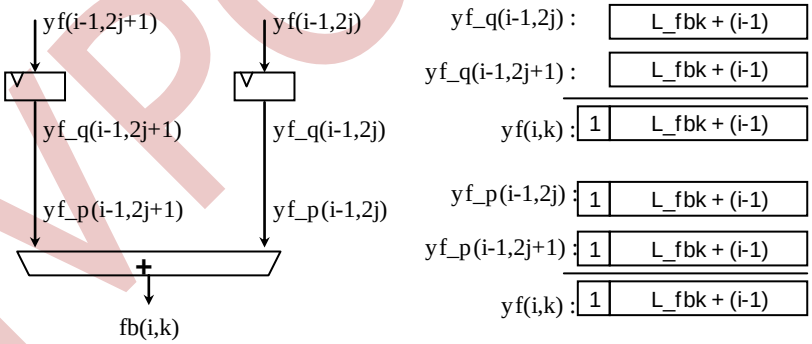

Figure 27. Length of 'yf_p', 'yf', 'yf_q' signals

It is easy to conclude that: $\boxed{Len\{yf(i,:)\} = L\_fbk + i}$

- Recall that the maximum number of bits needed is L_FIR. The smaller summations are also bounded by L_FIR. In this implementation, we truncate the number of bits to L_FIR at the last level (unlike the previous FIR block case where we could truncate at any level with a bit-width larger than L_fbk). The reason is that the bit-width growth is not as pronounced as in the FIR block case.

SATURATION/TRUNCATION SCHEME

Recall:

- OP: Output truncation scheme:
  - OP = 0: A truncation is performed at the LSBs, and then saturation is performed, so that the output fixed-point format results [NO NQ].
  - OP = 1: A truncation is performed at both the LSB and MSB parts, so that the output fixed-point format results [NO NQ].
  - OP = 2: The maximum number of allowable bits is used for the output format, which is given by L_FIR.
- [NO NQ]: Output fixed-point format (valid if OP ≠ 2). NO: number of integer bits. NQ: number of fractional bits. The user can choose the output format, provided is within the bounds of the maximum allowable number of bits for both the integer and fractional part. Note that:

$$NQ \le NO \le L\_FIR, \quad 0 \le NQ \le NH + B - 2$$

Figure 28 shows the bit structure after saturation/truncation. The final output 'y' fixed-point representation is L_FIR bits. It stays like that only in OP = 2. If OP ≠ 2, we perform LSB truncation and MSB Saturation (OP = 0) or truncation (OP = 1). We also located the fractional bit where it should be.

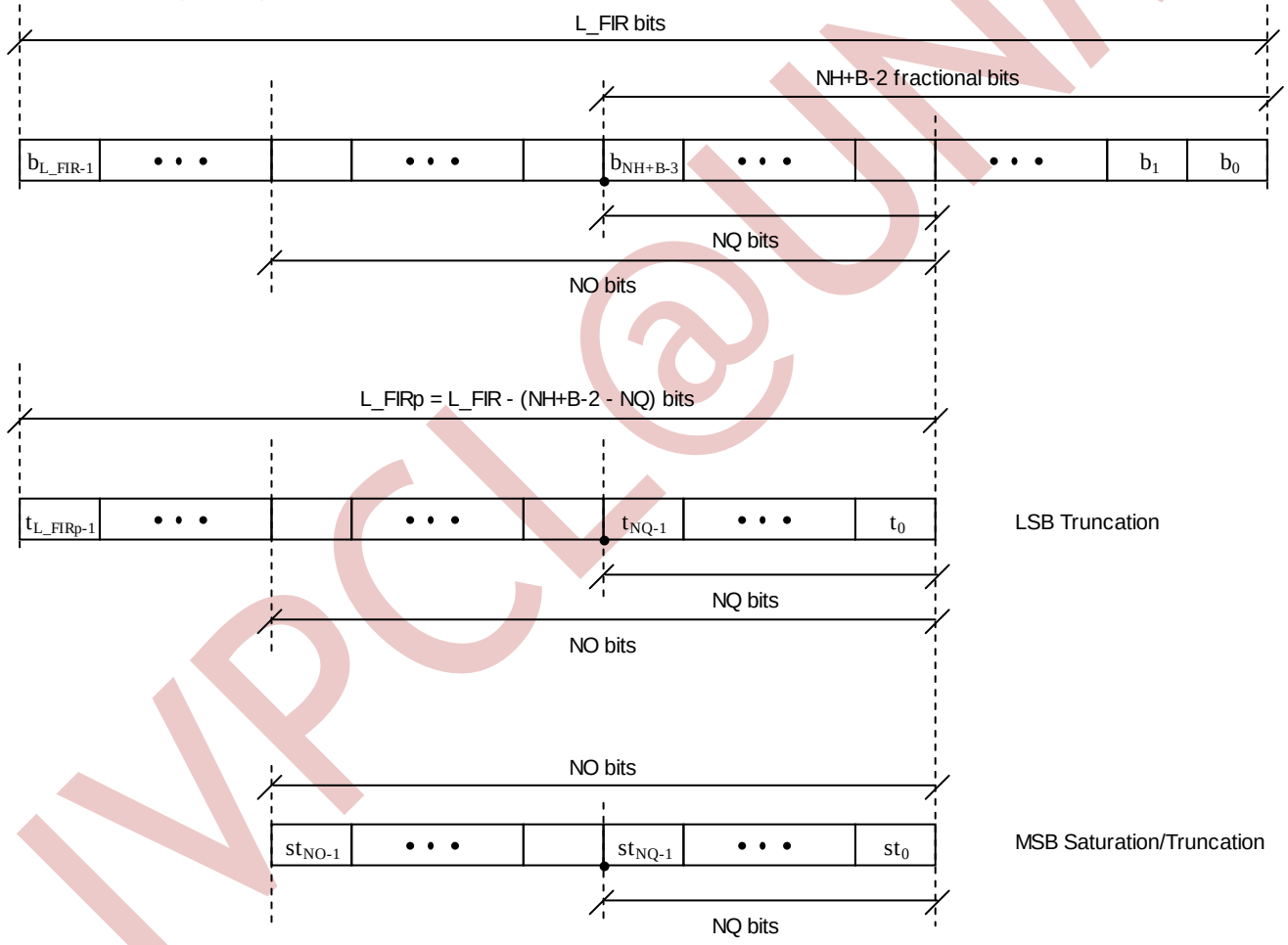

Figure 28. Saturation/Truncation scheme

If OP ≠ 2, we want to get the new format [NO NQ]. LSB Truncation can be easily accomplished as shown in Fig. 28, there we get rid of the rightmost bits, so that only 'NQ' fractional bits remain.

The next operation (either MSB Saturation or Truncation) depends on the parameter OP.

- OP = 1: Truncation at the MSB part and LSB parts. Here, we only need to get rid of a number of leftmost bits so that only 'NO-NQ' integer bits remain (shown in Fig. 28).

- OP = 0: Saturation at the MSB part and truncation at the LSB part. This case requires special attention, since three cases arise. Those cases are shown in Figure 29.
  i. All the 'L_FIRp-NO+1' bits are identical ('1' or '0'): Here, truncate to 'NO' bits.
  ii. Rightmost bit = '1', and the other 'L_FIRp-NO+1' bits are not all equal to '1': Here, saturate to the largest negative value, i.e. -2^{NO-1}.
  iii. The first bit is '0', and the other 'L_FIRp-NO+1' bits are not all equal to '0': Here, saturate to the largest positive value, i.e. $2^{NO-1}-1$.

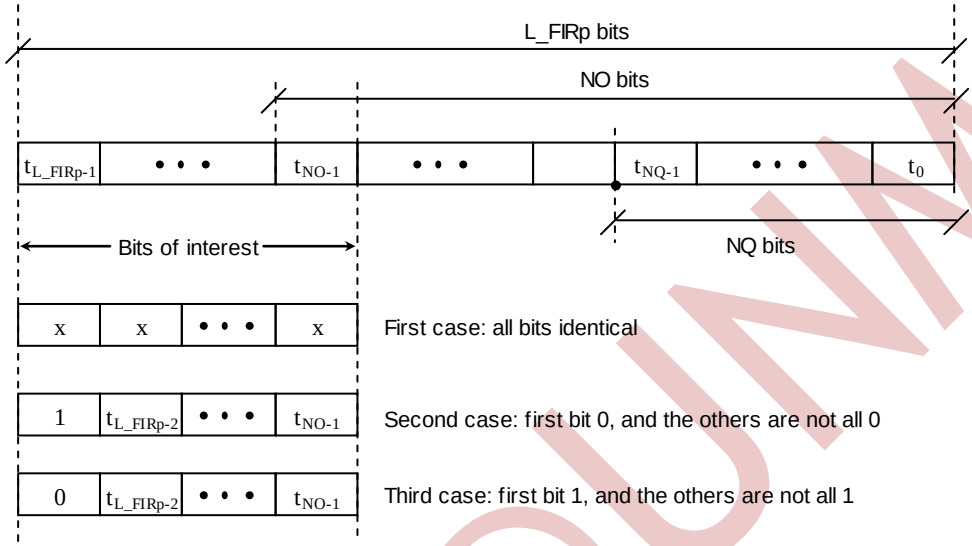

Figure 29. Three cases that arise when OP = 0

Keep in mind that the largest negative number and largest positive number described above are in integer representation. It would have been correct if we used its fixed-point representation with [NO NQ] format, but the equation would have been more complex, and it is not worth the effort.
So, when saturating to the largest positive or negative value, pretend for a moment the fractional point does not exist; once the value has been written, put back the fractional point to its original position.