HEVC Transform, Scaling, Quantization

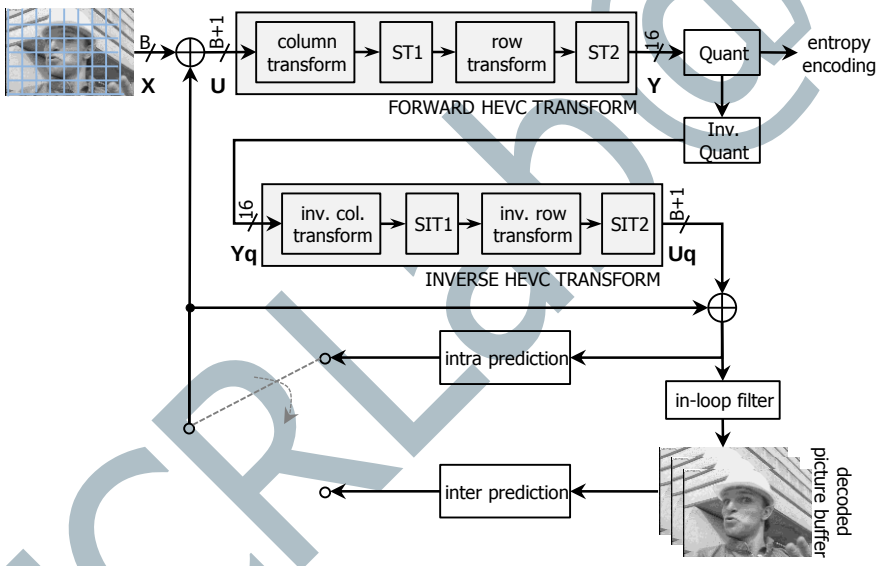
HEVC

EQUATIONS

- Forward HEVC Transform and Scaling: U = residual image with $B+1$ bits.
  - ✓ 1st Transform: $D \times U$
  - ✓ 1st Scaling: $Z = (D \times U) \times ST_1$. Bits per pixel of Z: 16 bits.
  - ✓ 2nd Transform: $Z \times D^T$
  - ✓ 2nd Scaling: $Y = (Z \times D^T) \times ST_2$. Bits per pixel of Y: 16 bits.

- Quantization: $Quant = Q(Y)$
- Inverse Quantization: $Yq = Dequant = Q^{-1}(Q(Y))$

- Inverse HEVC Transform and Scaling:
  - ✓ 1st Transform: $D^T \times Yq$
  - ✓ 1st Scaling: $Z = (D^T \times Yq) \times SIT_1$. Bits per pixel of Z: 16 bits.
  - ✓ 2nd Transform: $W \times D$
  - ✓ 2nd Scaling: $Uq = (W \times D) \times SIT_2$. Bits per pixel of Y: $B+1$ bits.



IMPLEMENTATION USING RECURSIVE EVEN-ODD DECOMPOSITION

- Forward Transform (1D row and column transforms): We first partition $D(N)$ into $D(N/2)$ and $M(N/2)$. Then, we partition $D(N/2)$ into $D(N/4)$ and $M(N/4)$. And so on. Note that the matrices $M(N/2^i)$ cannot be decomposed.
- Inverse transform (both 1D row and column transforms): We first partition $D^T(N)$ into $D^T(N/2)$ and $M^T(N/2)$. Then, we partition $D^T(N/2)$ into $D^T(N/4)$ and $M^T(N/4)$. And so on.
- We can also apply the symmetry/anti-symmetry property of $D(N)$ for efficient inner product implementation. This is functionally the same as non-recursive even-odd decomposition (even hardware resources are the same as we will also be using an add/sub structure but inside the inner products), though the coefficients are different. However, recursive even-odd decomposition is more efficient as we can keep decomposing the matrices, therefore requiring fewer resources.
- HEVC: We apply it to $N = 8,16,32$. The case $N = 4$ is not decomposed.
- Based on the reference algorithm:
  - ✓ The 1D Forward column and row Transforms require proper permutation of the input and output sequences as well as N pre-additions/pre-subtractions.
  - ✓ The 1D Inverse column and row Transforms require proper permutation of the input and output sequences as well as N post-additions/post-subtractions.

HEVC TRANSFORM AND SCALING

1D HEVC TRANSFORM - HARDWARE

This architecture (fully parallel, iterative) receives the input block column-wise and generates one output column at a time.

FULLY PARALLEL IMPLEMENTATION

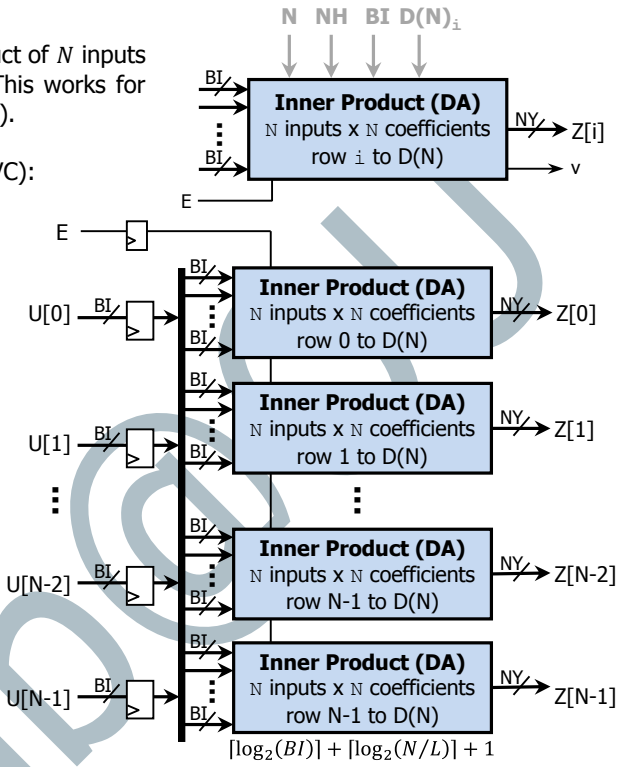
- The following figure shows the implementation of an inner product of N inputs by N coefficients (no symmetry or even-odd decomposition). This works for row and column transforms (or both Forward/Inverse Transform).

- In this case, the number of output bits is given by (NH=8 in HEVC):
$$NY = NH + BI + \lceil \log_2(N+1) \rceil - 1$$

- Residual image bitwidth: $BI = B + 1$. B: Input image bitwidth. This is true for the 1st Transform. For the 2nd transform $BI = NO = 16$.
- We can feed a new column every cycle. The I/O delay is:
$$\lceil \log_2 BI \rceil + \lceil \log_2 N/L \rceil + 1$$

- The following figure shows the implementation of a $N \times N$ transform (no symmetry or even-odd decomposition). For HEVC, with B=8, BI=9 for column Transform, BI=16 for row Transform.

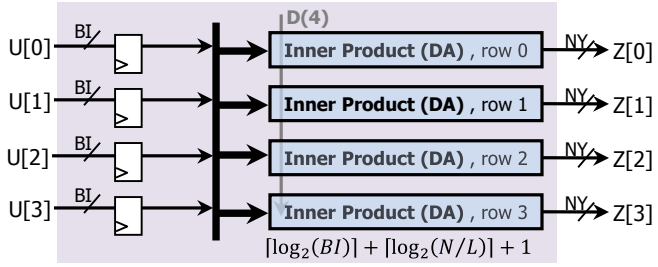
- We can feed a new column every cycle. The I/O delay is:
$$\lceil \log_2 BI \rceil + \lceil \log_2 N/L \rceil + 2$$



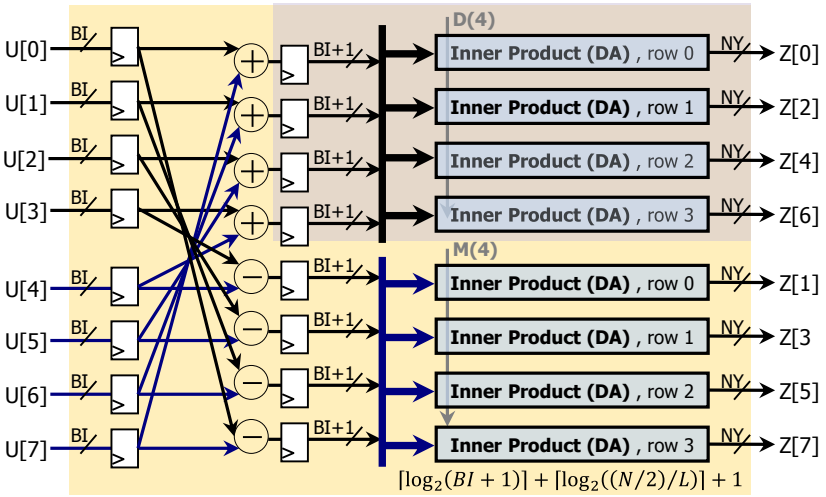
Recursive even-odd decomposition - Architecture: Forward Transform

- Signal E: It arrives at the same time data arrives at the input. In our figures, it is implied (not shown) that E is delayed the same amount as the input data to correctly arrive at the inner products. An alternative implementation (also called non-recursive even-odd decomposition) uses symmetry of $D(N)$ to decompose an $N \times N$ unit into two $N/2 \times N/2$ units. This works fine up to 8x8; after that, we better use recursive even-odd decomposition.

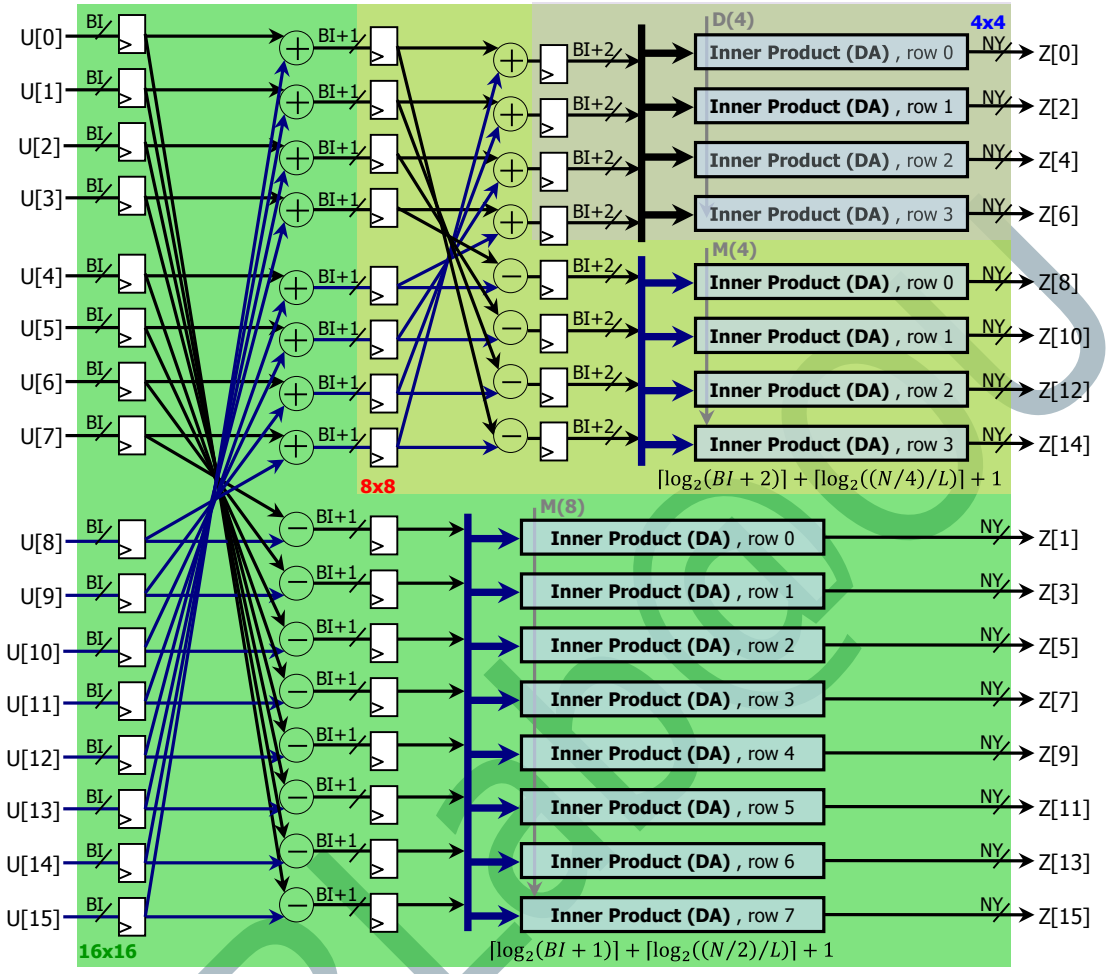
- 4x4: smallest case, no decomposition. The enable into the inner products includes one delay unit (not shown).



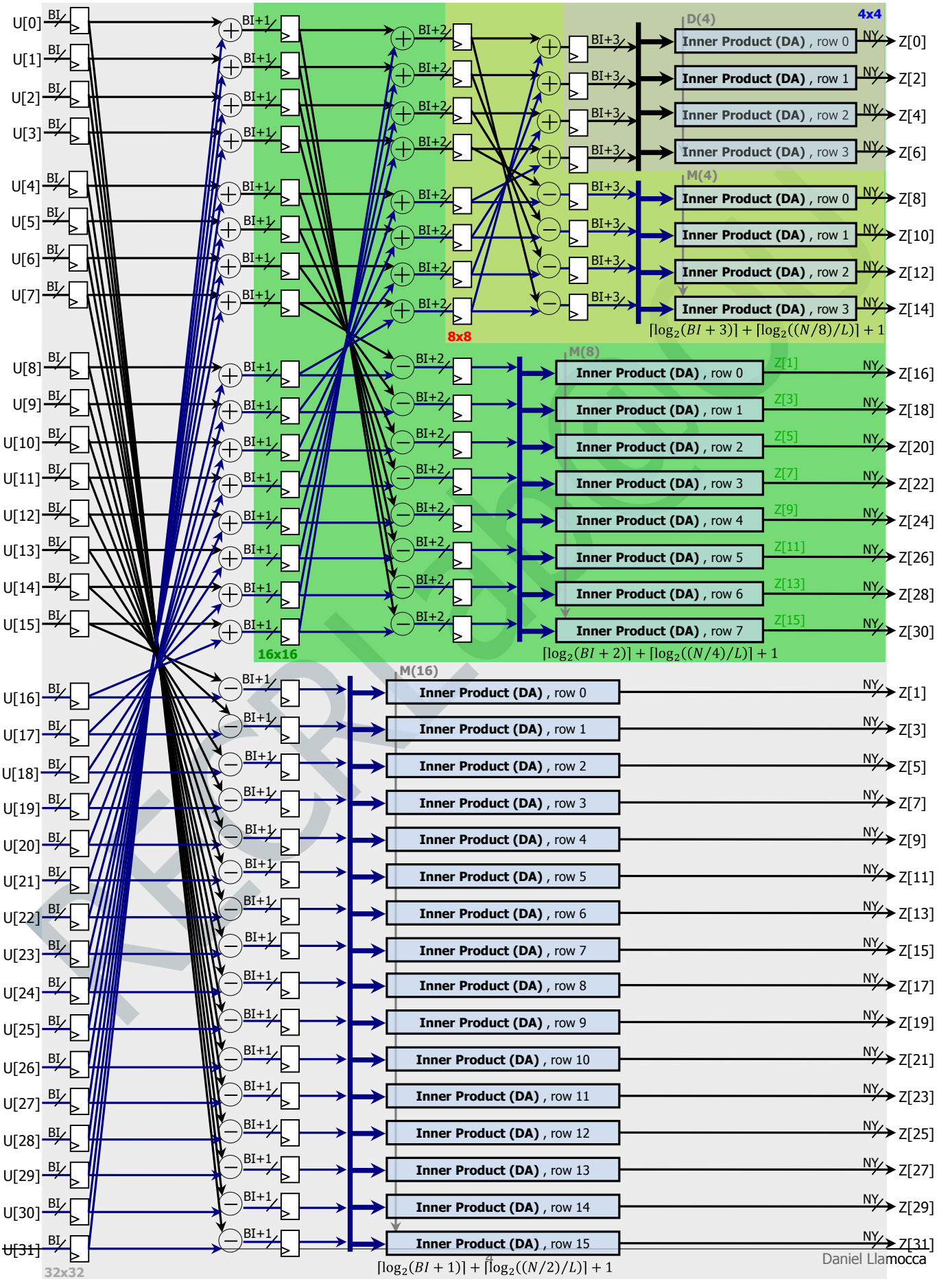
- 8x8: It is made of two 4x4 units. Top half: 4x4 unit (for $D(4)$). Note that both 4x4 units receive input data of $BI + 1$ bits. Note that due to the adder/subs, we include an extra register level (two delays for the enable input).



- 16x16: Here, the even-odd decomposition pays off (it is better than taking advantage of the symmetry of $D(16)$). Top Half: 8x8 unit for $D(8)$. Bottom Half: 8x8 unit for $M(8)$ (cannot be decomposed further). The enable for the inner products of $D(4)$ and $M(4)$ has 3 delay units; the enable for $M(8)$ units has 2 delay units.



- 32x32: Top half: 16x16 unit for D(16). Bottom half: 16x16 unit for M(16) (we cannot decompose it further). The enable for $D(4)$ and $M(4)$ has 4 delay units; the enable for $M(8)$ units has 3 delay units; the enable for $M(16)$ has 2 delay units.



I/O delay: (between an input column and its corresponding output column)

- No even-odd decomposition nor symmetry: $\lceil \log_2(BI) \rceil + \lceil \log_2(N/L) \rceil + 2$ cycles. Measured between input data ready ($E = 1$) and output data ready ($v = 1$). In practice, we use recursive even-odd decomposition.
- Recursive even-odd decomposition: The previous formula still applies to the non-decomposed portions. Note that the register the lower portions (that do $M(N/2)$) have more delay, this makes sure that all output data $Z(0)$ to $Z(N-1)$ (with valid v signals) appear at the same time. For example (L=4):

$N = 4$	4x4 unit (D(4)):			$\lceil \log_2(BI) \rceil + \lceil \log_2(N/L) \rceil + 1$
	Total time (1 register level)			$\lceil \log_2(BI) \rceil + \lceil \log_2(N/L) \rceil + 2$
$N = 8$	Bottom unit (M(4), not decomposed):			$\lceil \log_2(BI+1) \rceil + \lceil \log_2((N/2)/L) \rceil + 1$
	Top 4x4 unit (D(4)):			$\lceil \log_2(BI+1) \rceil + \lceil \log_2((N/2)/L) \rceil + 1$
	Total time (2 extra register level):			$\lceil \log_2(BI+1) \rceil + \lceil \log_2((N/2)/L) \rceil + 3$
$N = 16$	Bottom unit (M(8), not decomposed):			$\lceil \log_2(BI+1) \rceil + \lceil \log_2((N/2)/L) \rceil + 1$
	Top 8x8 unit (D(8))	Bottom unit (M(4), not decomposed):		$\lceil \log_2(BI+2) \rceil + \lceil \log_2((N/4)/L) \rceil + 1$
		Top unit (4x4, D(4)):		$\lceil \log_2(BI+2) \rceil + \lceil \log_2((N/4)/L) \rceil + 1$
	Total time (2 extra register levels):			$\lceil \log_2(BI+1) \rceil + \lceil \log_2((N/2)/L) \rceil + 3$
$N = 32$	Bottom unit (M(16), not decomposed):			$\lceil \log_2(BI+1) \rceil + \lceil \log_2((N/2)/L) \rceil + 1$
	Top 16x16 unit (D(16))	Bottom unit (M(8), not decomposed):		$\lceil \log_2(BI+2) \rceil + \lceil \log_2((N/4)/L) \rceil + 1$
		Top 8x8 unit (D(8))	Bottom 4x4 unit (M(4):	$\lceil \log_2(BI+3) \rceil + \lceil \log_2((N/8)/L) \rceil + 1$
			Top 4x4 unit (D(4)):	$\lceil \log_2(BI+3) \rceil + \lceil \log_2((N/8)/L) \rceil + 1$
	Total time (2 extra register levels):			$\lceil \log_2(BI+1) \rceil + \lceil \log_2((N/2)/L) \rceil + 3$

The balance between the inner products and the register levels works as long as $\lceil \log_2(BI+1) \rceil = \lceil \log_2(BI+2) \rceil = \lceil \log_2(BI+3) \rceil$. This is true for $BI = 9{,}16$.
In general, the formula (for N power of 2, L=4, using the smallest unit and then adding the register levels) is:
$\lceil \log_2(BI + \lceil \log_2(N/4) \rceil) \rceil + \lceil \log_2((N/(N/4))/L) \rceil + 1 + 1 + \lceil \log_2(N/4) \rceil = \lceil \log_2(BI + \lceil \log_2(N/4) \rceil) \rceil + 2 + \lceil \log_2(N/4) \rceil$

- ✓ How fast can we feed a new column? Every clock cycle.

Maximum number of output bits:

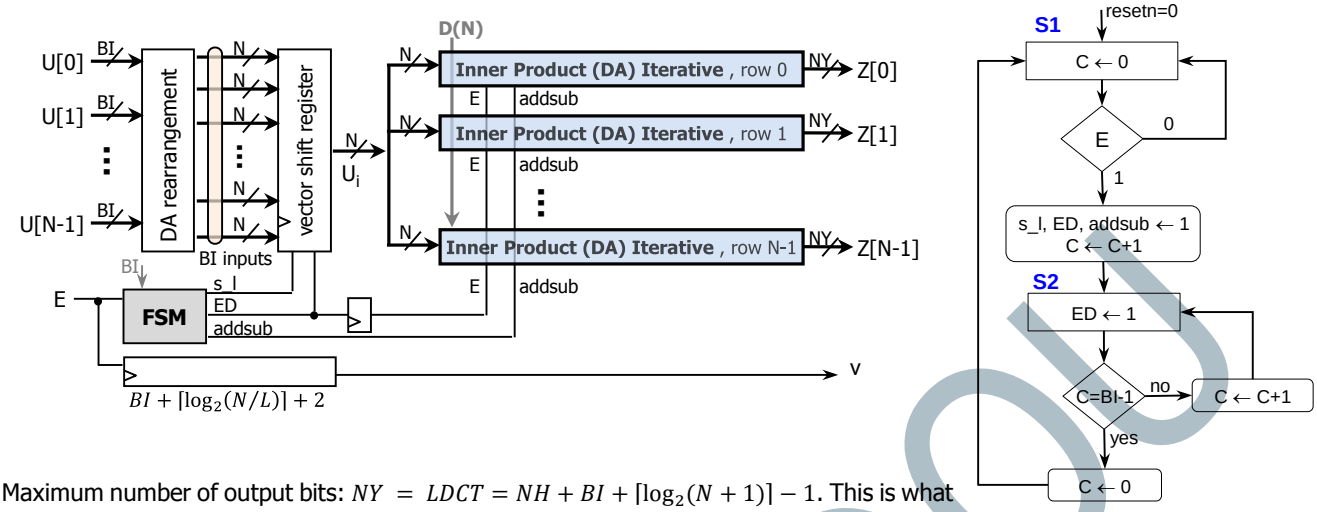
- This is what a generic hardware would give us. We will find out that for HEVC, this is too many bits, and we will chop off accordingly (but this is later, this is not Scaling):

$N = 4$	D(4):			$NH + BI + \lceil \log_2(N+1) \rceil - 1$
$N = 8$	Top 4x4 unit (D(4)):			$NH + BI + 1 + \lceil \log_2(N/2+1) \rceil - 1$
	Bottom 4x4 unit (M(4)):			$NH + BI + 1 + \lceil \log_2(N/2+1) \rceil - 1$
$N = 16$	Top 8x8 unit (D(8))	Top unit: D(4)		$NH + BI + 2 + \lceil \log_2(N/4+1) \rceil - 1$
		Bottom unit: M(4)		$NH + BI + 2 + \lceil \log_2(N/4+1) \rceil - 1$
	Bottom unit: M(8)			$NH + BI + 1 + \lceil \log_2(N/2+1) \rceil - 1$
$N = 32$	Top 16x16 unit (D(16))	Bottom unit: M(8):		$NH + BI + 2 + \lceil \log_2(N/4+1) \rceil - 1$
		Top 8x8 unit (D(8))	Top 4x4 unit (D(4):	$NH + BI + 3 + \lceil \log_2(N/8+1) \rceil - 1$
			Bottom 4x4 unit (M(4)):	$NH + BI + 3 + \lceil \log_2(N/8+1) \rceil - 1$
	Bottom unit: M(16):			$NH + BI + 1 + \lceil \log_2(N/2+1) \rceil - 1$

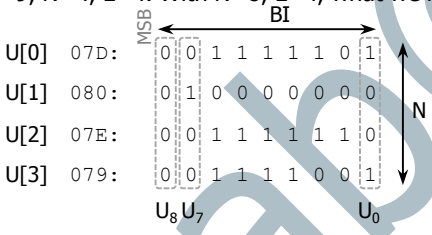
- All these formulas are the same, as: $1 + \lceil \log_2(N/2+1) \rceil = \lceil \log_2(N+1) \rceil$ for N=4, 8, 16, 32.

ITERATIVE IMPLEMENTATION

- The following figure shows the implementation of a $N \times N$ transform (no symmetry or even-odd decomposition):

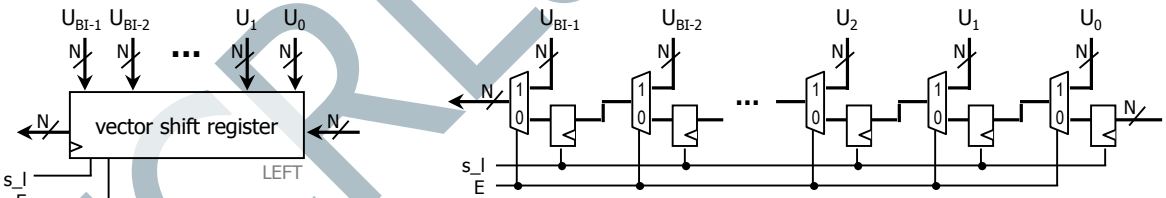


- Maximum number of output bits: $NY = LDCT = NH + BI + \lceil \log_2(N+1) \rceil - 1$. This is what is required by simple analysis. However, we will find that when doing specialized analysis, we actually require fewer bits. So, we will chop off bits from the MSB. Moreover, there is a step called scaling (getting rid of LSBs to get 16 output bits).
- Each column in a block can be entered every BI cycles (if BI=1, we input a new column every cycle).
- DA rearrangement: example with BI=9, N=4, L=4. With N=8, L=4, what we have is 2 groups of U8-U0 (see FIR DA Report).



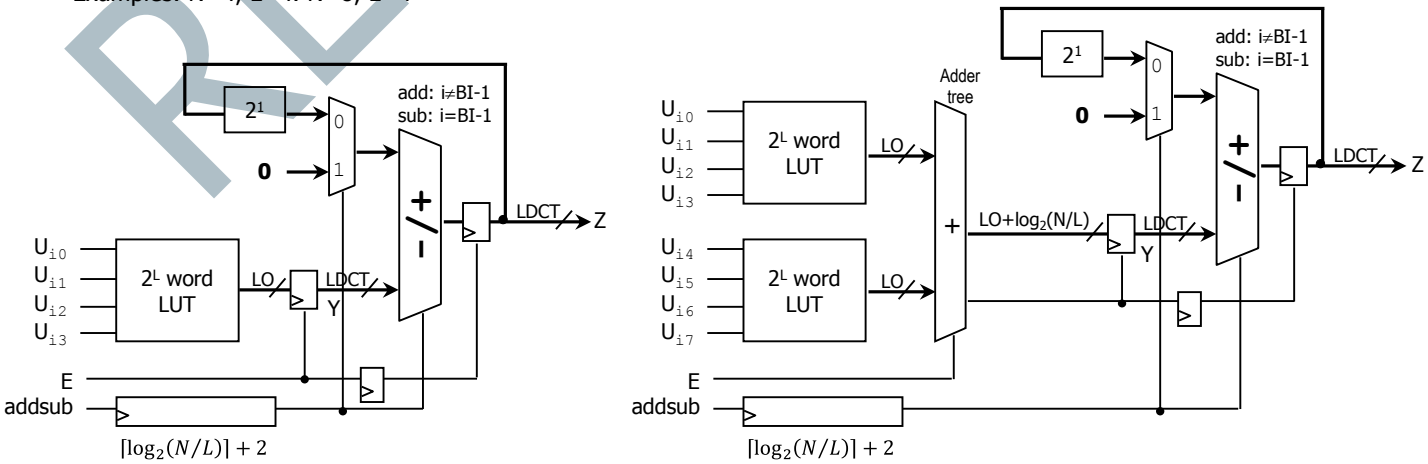
Vector Shift Register

- This circuit receives N BI-bits inputs and then shifts out each N-bit group using s_I and E. When s_I=E=1, we load parallel data. When s_I=0, E=1, we shift to the left.



Inner Product

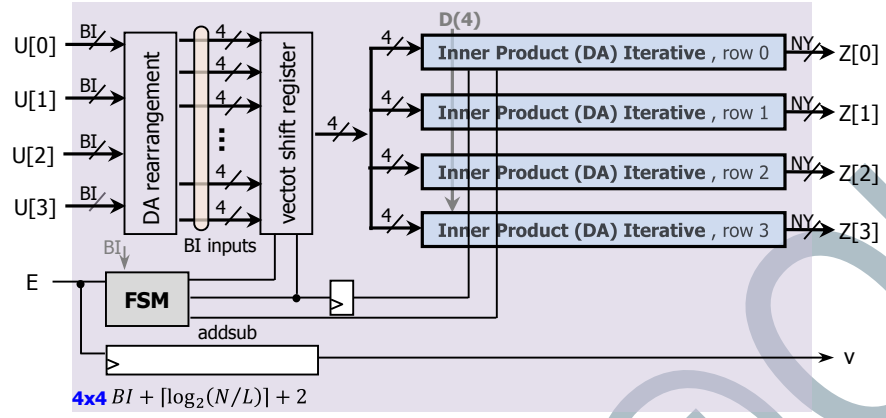
- Input Parameters: N, L, NH, BI, NO, and the LUT file. We use op=2 (largest output format)
- Examples: N=4, L=4. N=8, L=4



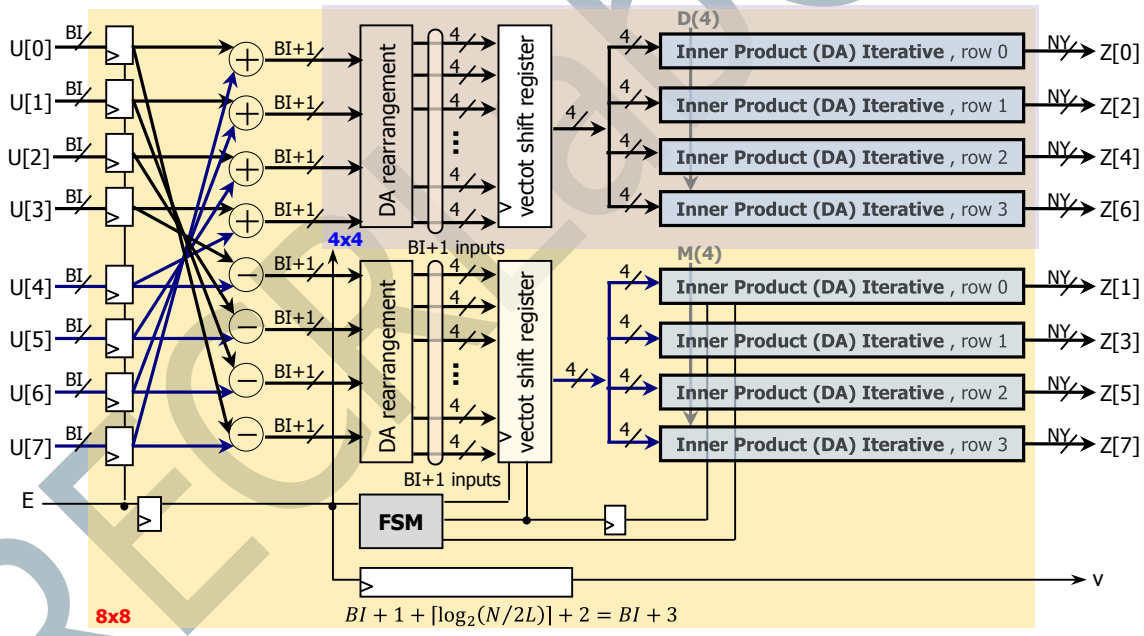
Recursive even-odd decomposition - Architecture: Forward Transform

- Signal 'E' (enable to the FSM): It has to arrive at the same time data arrives on the vector shift register. Note that for 16x16 and 32x32, we need to include register levels; this increases the execution time and the time between assertions of input enable 'E' (as compared to the non-recursive even-odd decomposition).

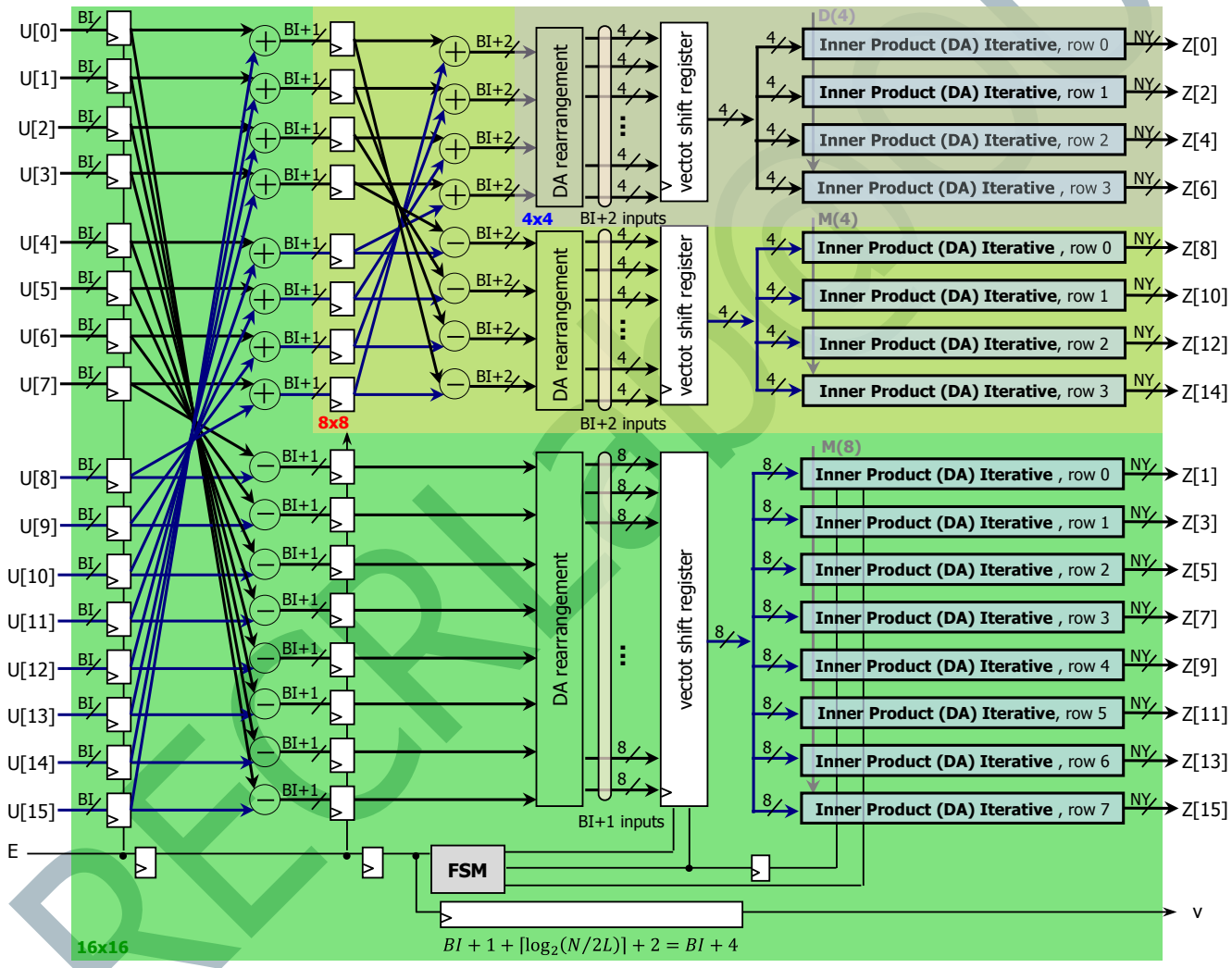
- 4x4: smallest case, no decomposition. Note that the input data (BI bits per group) to the DA rearrangement block does not require input registers (because the DA rearrangement block is only signal rearrangement).



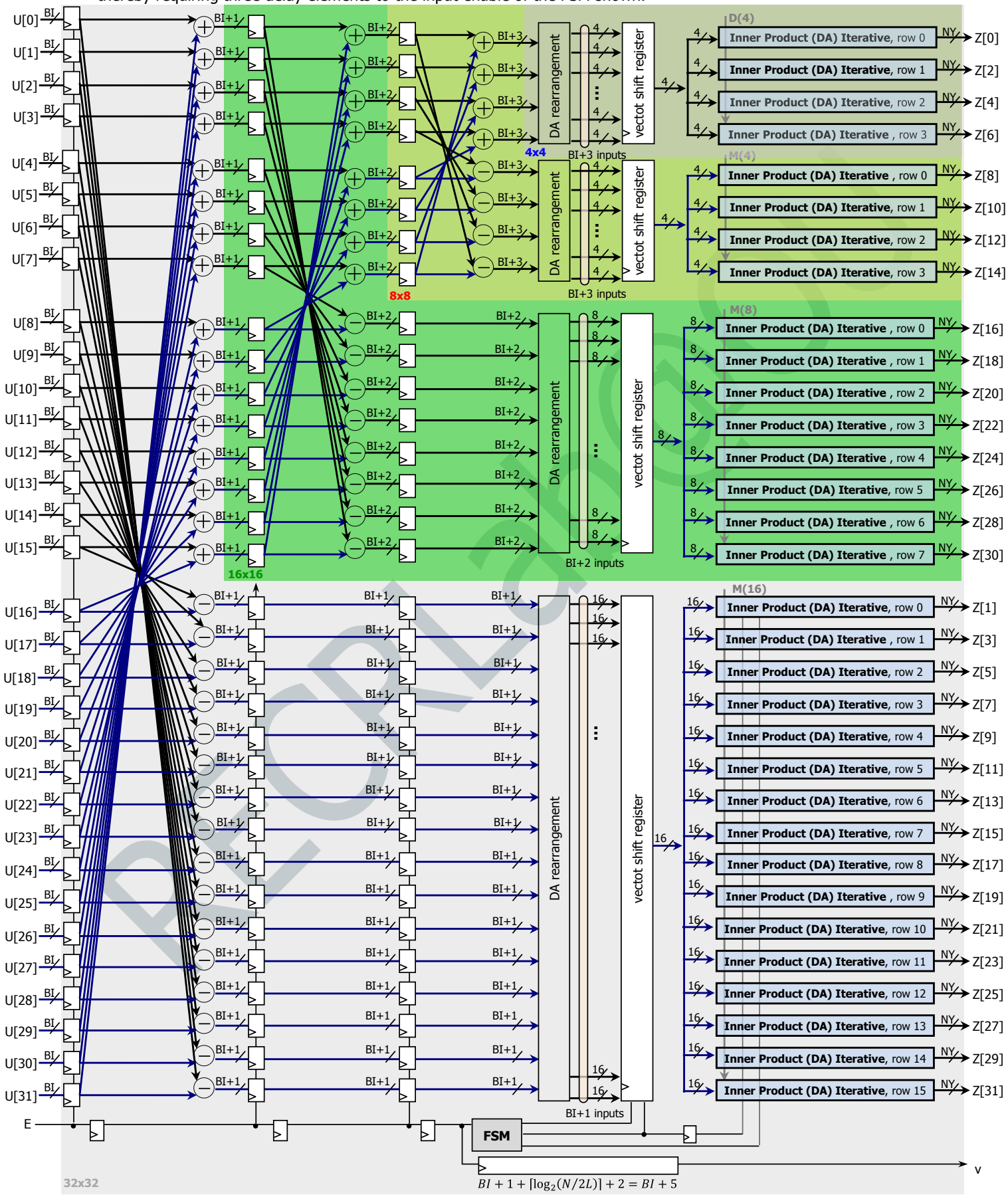
- 8x8: It is made of two 4x4 units. Top half: 4x4 unit for $D(4)$ with its own FSM (not shown in the figure). A simpler circuit would have been to have only one FSM to control both units (and using D(8) with symmetry), but we keep it this way for consistency with the even-odd decomposition. Note that both 4x4 units receive input data of $BI + 1$ bits.
Note that due to the adder/subs, we register the inputs, adding a delay element to the input enable of the FSM shown.



- 16x16: Here, the even-odd decomposition pays off (it is better than taking advantage of the symmetry of D(16)). The top half is a 8x8 unit, while the bottom is not decomposed at all (it uses M(8)).
  - ✓ Because of the two levels of adder/subs to get to feed data to the top 4x4 unit, we need two register levels for the inputs, thereby requiring two delay elements to the input enable of the FSM shown.
  - ✓ Note that if we had used non-recursive even odd decomposition (i.e., just symmetry), we could have saved one register level (and it would have run a bit faster). However, this circuit is more efficient in terms of resources.
  - ✓ Note that all the Iterative Inner Products take the same time, even when the input bits are different. This is because this balances out with the number of inputs to the vector shift registers. Note that the number of cycles that the vectors shift register and their inner products take is given by $BI + \log_2 N/L + 2$, where N is the number of inputs of the DA rearrangement unit, L=4 and BI the number of input bits to the DA rearrangement units. For example:
    - Vector shift register and inner products processing 8 bits: $BI + 1$ input bits to the DA rearrangement unit, $N = 8$. Then, $BI + 1 + \log_2 8/4 + 2 = BI + 4$ cycles.
    - Vector shift register and inner products processing 4 bits: $BI + 2$ input bits to the DA rearrangement unit, $N = 4$. Then, $BI + 2 + \log_2 4/4 + 2 = BI + 4$ cycles.



- 32x32: The top half is a 16x16 unit, while the bottom is not decomposed at all (it uses M(16)).
  Because of the three levels of adder/subs to get to feed data to the top 4x4 unit, we need three register levels for the inputs, thereby requiring three delay elements to the input enable of the FSM shown.



I/O delay: (between an input column and its corresponding output column)

- No even-odd decomposition nor symmetry: $BI + \lceil \log_2(N/L) \rceil + 2$ cycles. We load BI N-bit inputs, and shift them all out to the inner product. Measured between input data ready ($E = 1$) at the DA rearrangement block and output data ready ($v = 1$). In practice, we use recursive even-odd decomposition. To be fair, no eo decomposition is faster, but more resources!
- Recursive even-odd decomposition: The formula applies only to the data coming into a DA rearrangement block (N': number of inputs, BI': number of input bits). Note that the register levels introduced into the lower portions (that do $M(N/2)$) make sure that all output data $Z(0)$ to $Z(N-1)$ (with valid v signals) appear at the same time. For example (L=4):

$N = 4$	Total time:			$BI + \lceil \log_2(N/L) \rceil + 2 = BI + 2$
$N = 8$	Bottom unit (M(4), not decomposed):			$BI + 1 + \lceil \log_2((N/2)/L) \rceil + 2 = BI + 3$
	Top 4x4 unit (D(4)):			$BI + 1 + \lceil \log_2((N/2)/L) \rceil + 2 = BI + 3$
	Total time (1 extra register level):			$BI + 3 + 1 = BI + 4$
$N = 16$	Bottom unit (M(8), not decomposed):			$BI + 1 + \lceil \log_2((N/2)/L) \rceil + 2 = BI + 4$
	Top 8x8 unit (D(8))	Bottom unit (M(4), not decomposed):		$BI + 2 + \lceil \log_2((N/4)/L) \rceil + 2 = BI + 4$
		Top unit (4x4, D(4)):		$BI + 2 + \lceil \log_2((N/4)/L) \rceil + 2 = BI + 4$
	Total time (2 extra register levels):			$BI + 4 + 2 = BI + 6$
$N = 32$	Bottom unit (M(16), not decomposed):			$BI + 1 + \lceil \log_2((N/2)/L) \rceil + 2 = BI + 5$
	Top 16x16 unit (D(16))	Bottom unit (M(8), not decomposed):		$BI + 2 + \lceil \log_2((N/4)/L) \rceil + 2 = BI + 5$
		Top 8x8 unit (D(8))	Bottom 4x4 unit (M(4):	$BI + 3 + \lceil \log_2((N/8)/L) \rceil + 2 = BI + 5$
			Top 4x4 unit (D(4)):	$BI + 3 + \lceil \log_2((N/8)/L) \rceil + 2 = BI + 5$
	Total time (3 extra register levels):			$BI + 5 + 3 = BI + 8$

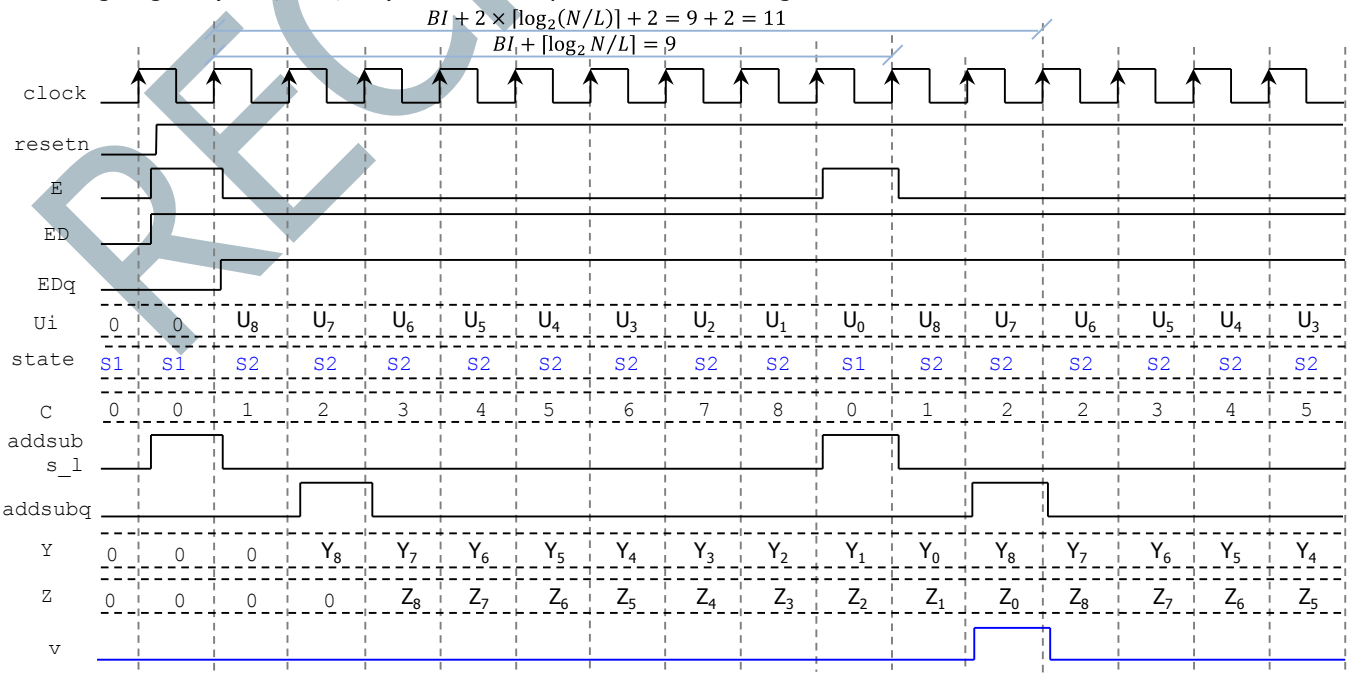
In general, the formula (for N power of 2) is: $BI + 2 \times \lceil \log_2(N/L) \rceil + 2$.

✓ How fast can we feed a new column? The vector shift register has to shift out as many inputs as it gets:

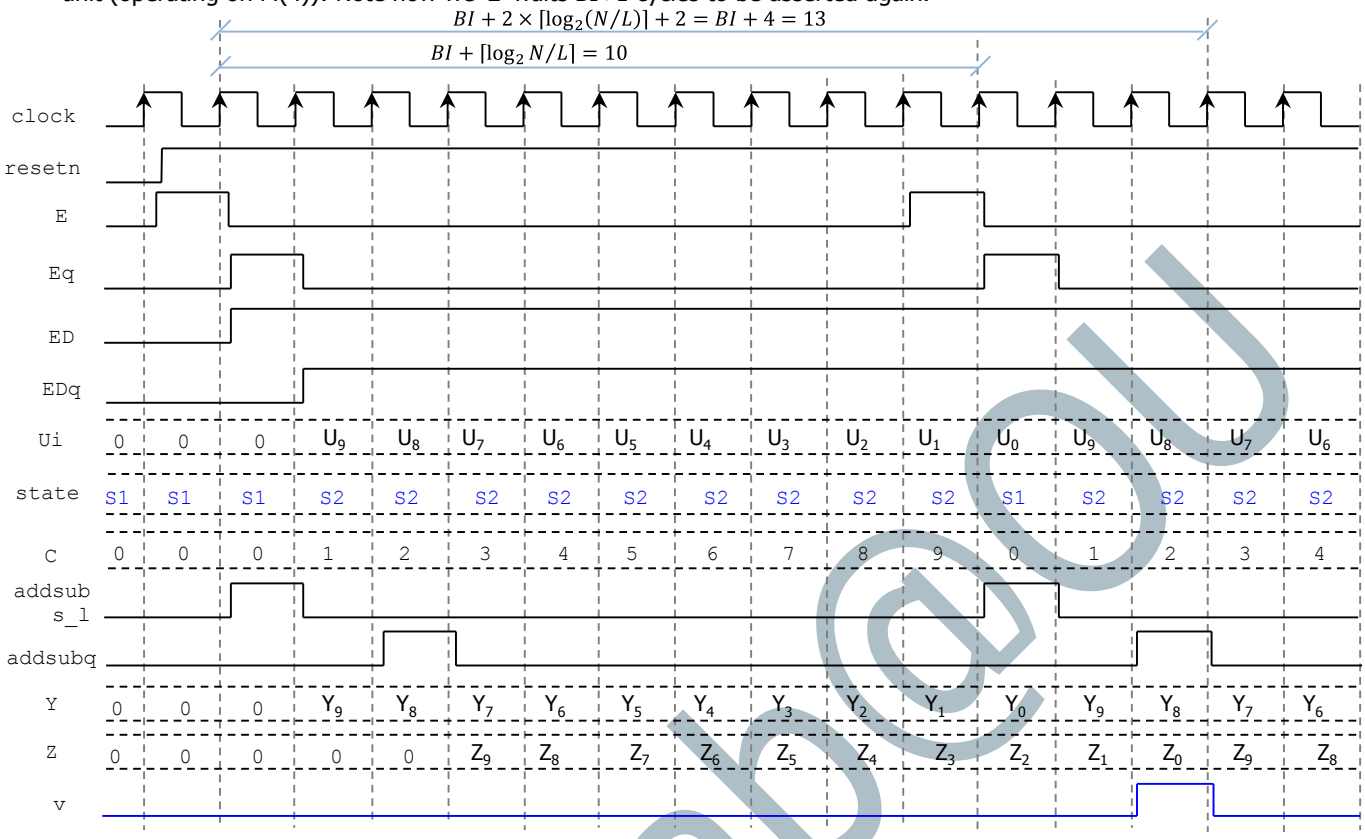
				Input bits	Time between assertions of 'E'
$N = 4$	Entire unit:			BI	BI cycles
$N = 8$	Entire unit:			BI	$BI + 1$ cycles
	Bottom unit (not decomposed):			$BI + 1$	
	Top unit (4x4):			$BI + 1$	
$N = 16$	Entire unit			BI	$BI + 2$ cycles
	Bottom unit (not decomposed)			$BI + 1$	
	Top half (8x8)	Bottom half:		$BI + 2$	
		Top half (4x4):		$BI + 2$	
$N = 32$	Entire unit:			BI	$BI + 3$ cycles
	Bottom unit (not decomposed):			$BI + 1$	
	Top 16x16 unit	Bottom half:		$BI + 2$	
		Top half (8x8)	Bottom half:	$BI + 3$	
			Top half (4x4):	$BI + 3$	

In general, the formula (for N power of 2) is: $BI + \lceil \log_2(N/L) \rceil$.

- Timing diagram (BI=9, N=4,L=4): 'E' waits BI cycles to be asserted again.



- Timing diagram (BI=9, N=8,L=4, even-odd decomposition): Here, the FSM signals correspond to the FSM of the bottom 4x4 unit (operating on M(4)): Note how we 'E' waits BI+1 cycles to be asserted again.



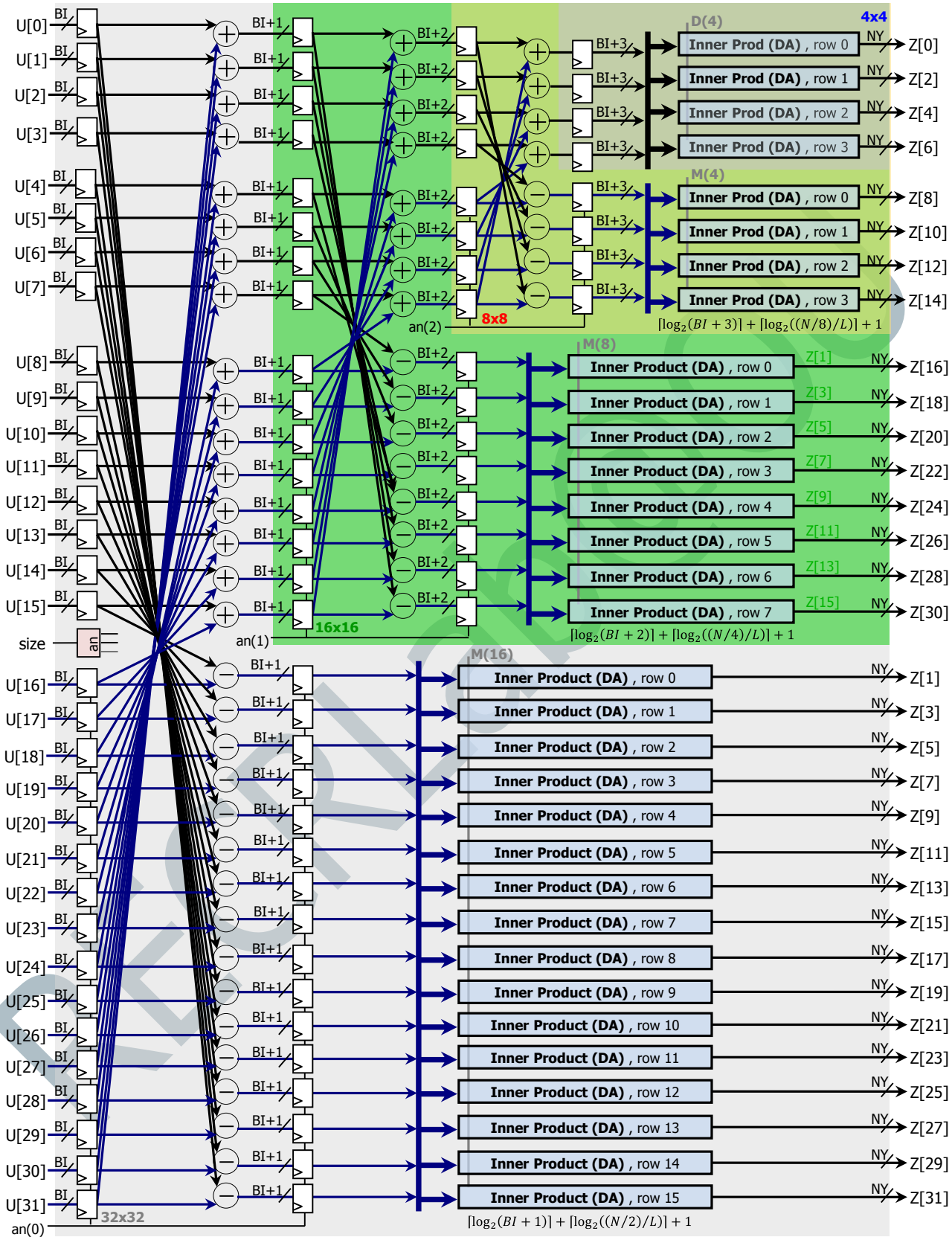
Computing HEVC Transform of different sizes

- Changing the size of the Transform would require run-time alteration of the architecture to achieve the desired size.
- However, in HEVC, the block sizes inside a CTB vary continuously. So, the overhead of run-time alteration would offset any benefit it might have had.
- A better approach is to control the synchronous clear ($an(2..0)$) where we feed zeros to guarantee proper results when we require smaller transform than N.
- For example, a 32x32 core can process any of 4x4, 8x8, 16x16, and 32x32 Transforms. We use the input $size$ and a decoder to get the signal $an(2..0)$ (we can omit the input $size$ and just have $an(2..0)$ as the input). The only drawback is that the processing time of all transform sizes will be that of the 32x32 case (the circuitry is still the transform hardware for 32x32).

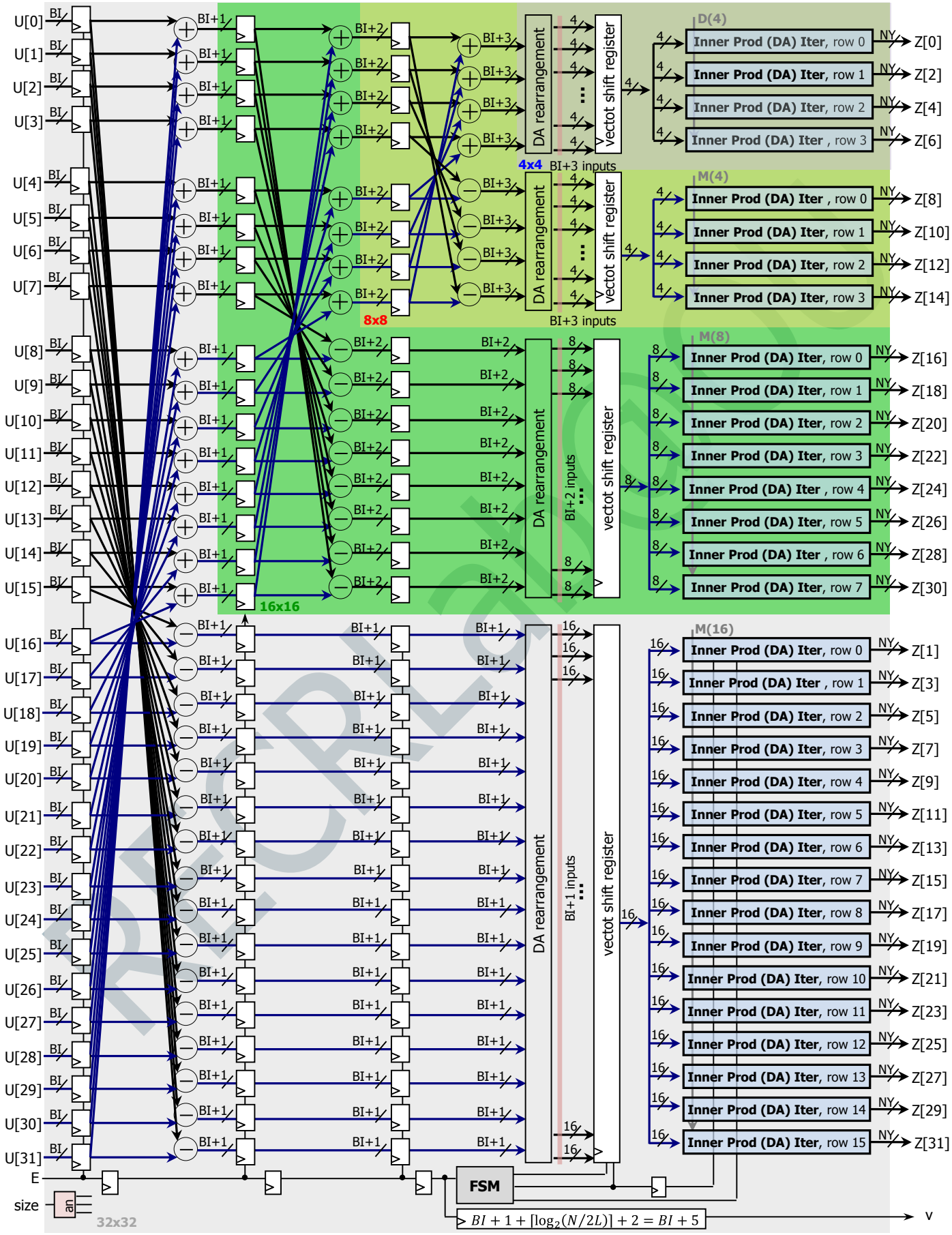
$an(2..0)$	Transform	$size$
111	4x4	00
011	8x8	01
001	16x16	10
000	32x32	11

- Now, we present 32x32 hardware architectures that compute cores of different sizes via the input size of $an(2..0)$.

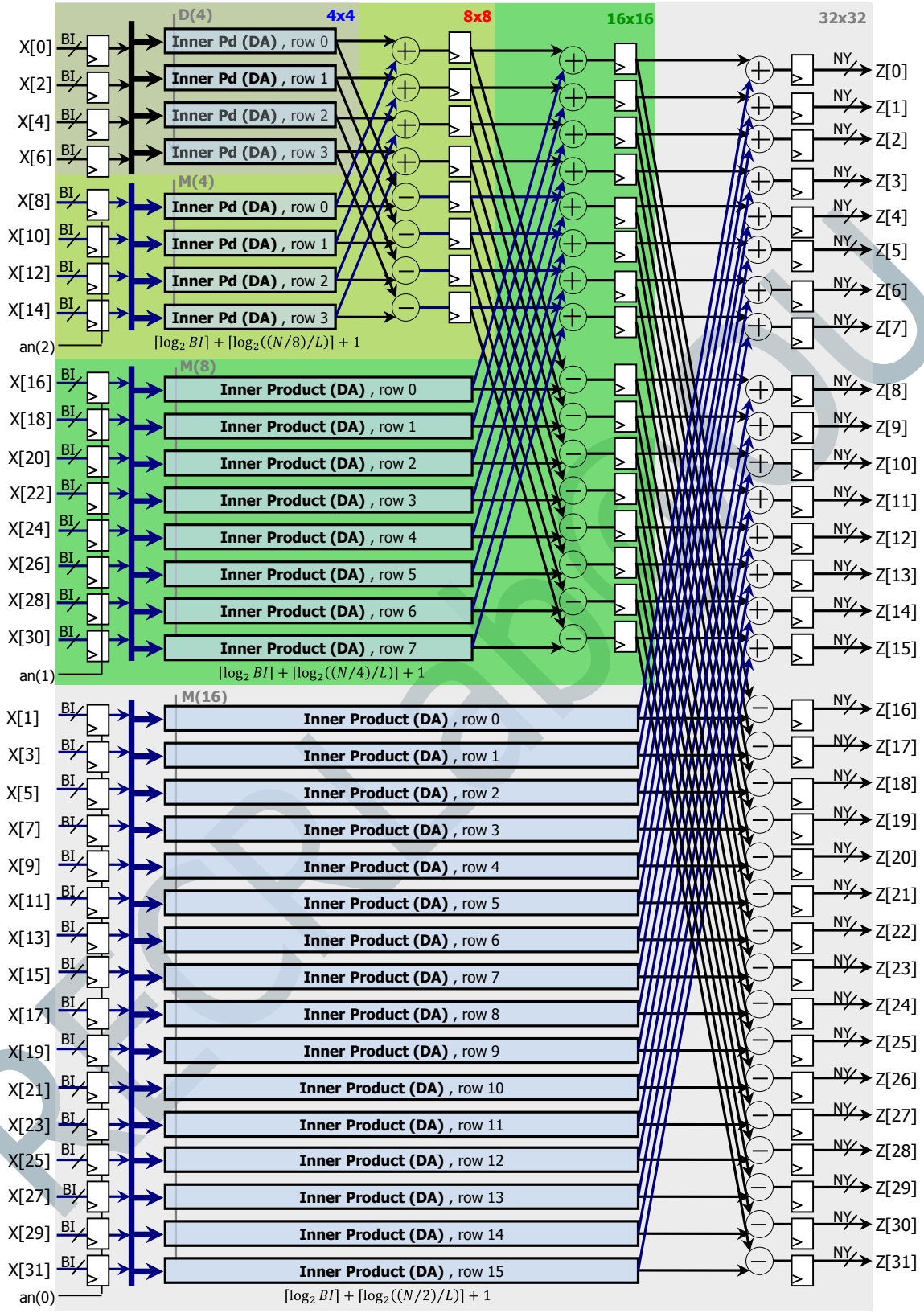
- Forward Transform: 32x32, fully parallel.



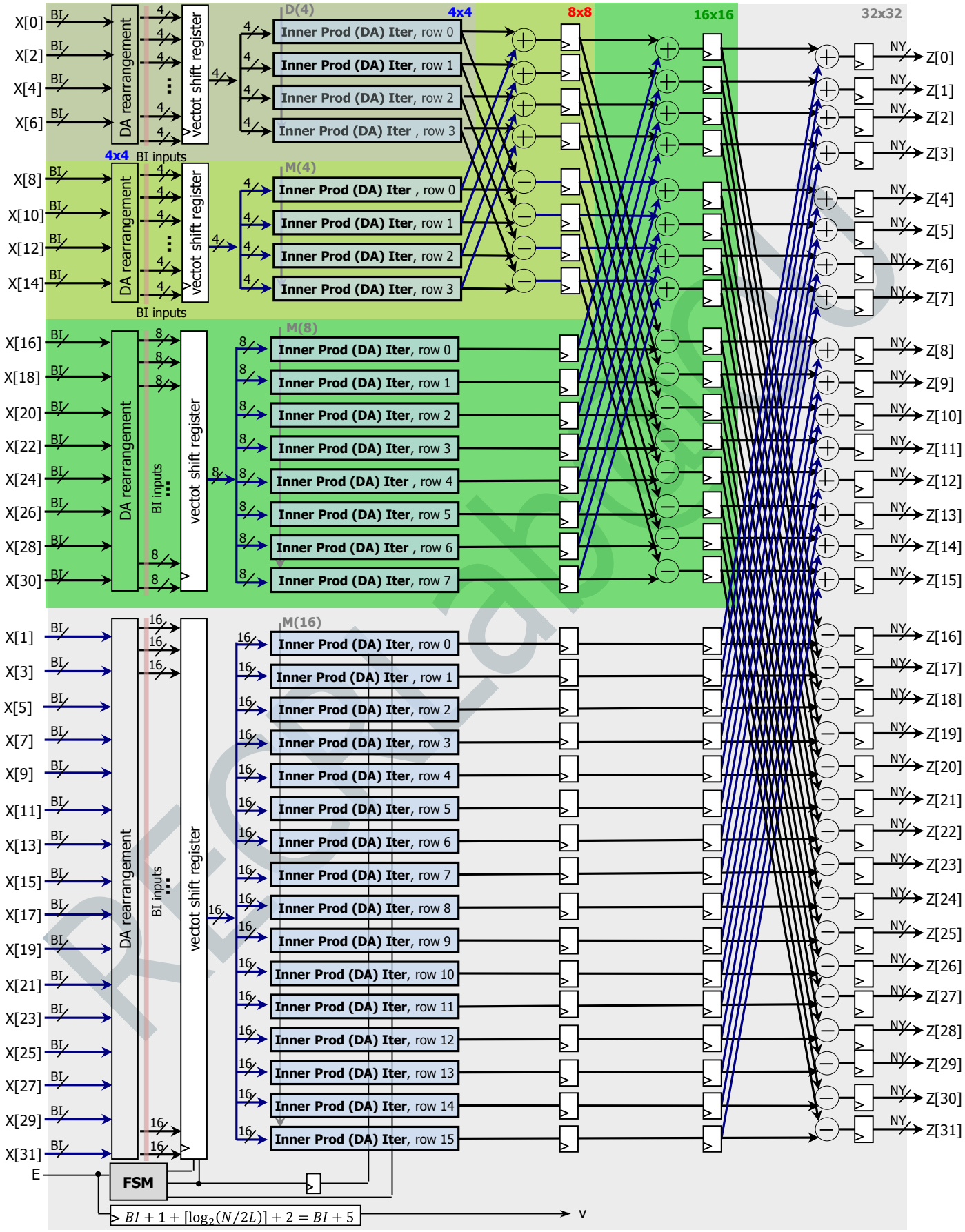
- Forward Transform: 32x32, iterative.



- Inverse Transform: 32x32, fully parallel. Note how the adder and subtractors operate on the output of the inner products.



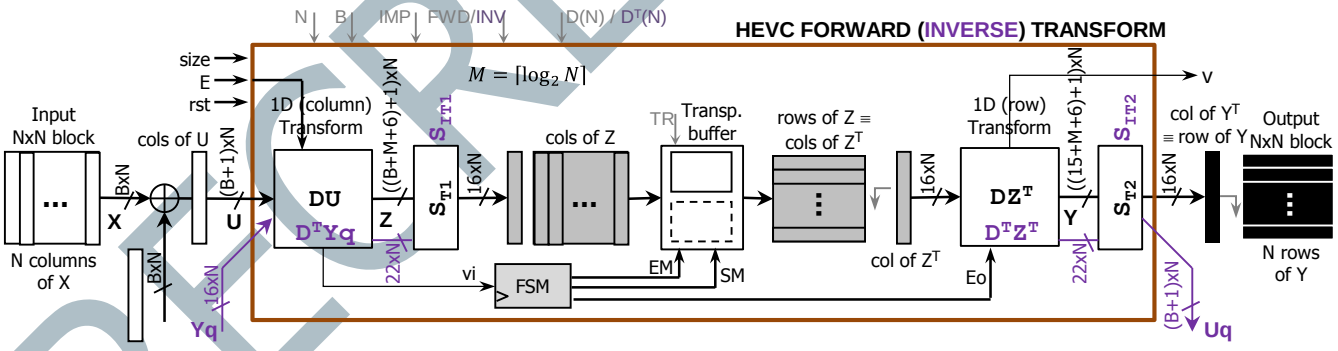
- Inverse Transform: 32x32, iterative.



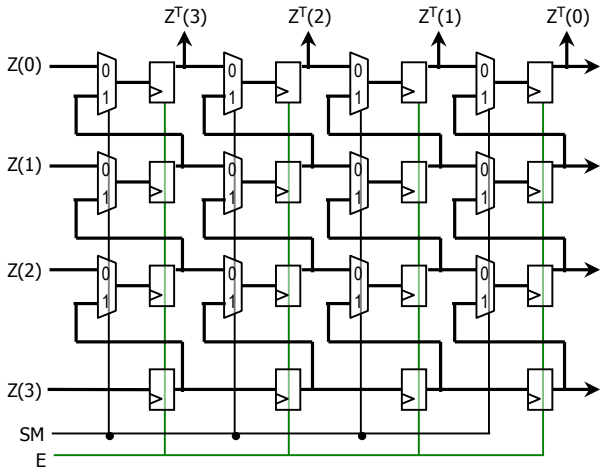
2D HEVC TRANSFORM AND SCALING - HARDWARE

- X: Input image, B bits per pixel.
- U: Residual image, $BI = B + 1$ bits per pixel, $U \in [-2^B + 1, 2^B - 1]$. For 2nd Transform, $BI = NO = 16$.
- Implementation details. We use the HEVC Forward Transform here, but something similar applies to the Inverse Transform.
  - ✓ For clearness, we use the same signal name before and after any Scaling operation (since this is a trivial hardware operation: bit pruning). For example, we use Z to name the signal before and after S_{T1} Scaling.
  - ✓ 1D transform architecture: It computes $C \times P$, where the input block P (fed column-wise) is multiplied by a constant matrix C. The output block is generated column-wise.
  - ✓ **1D Column Transform**: It receives block U and computes $Z = DU$. This maps exactly to our 1D transform architecture when U is provided column-wise; note that Z is generated column-wise.
    - S_{T1}: HEVC requires each pixel of Z to be pruned to 16 bits. Thus, we have to scale the outputs of the first 1D Transform so that they require at most 16 bits: $-2^{B+M+6} \times 2^x = -2^{15}$. Then $S_{T1} = 2^x = 2^{-(B+M-9)}$.
    - Maximum output pixel value: $-2^B \times N \times 64 = -2^{B+M+6}$ (we use -2^B instead of $-2^B + 1$ for a simpler analysis). Thus, the output Z needs $(B + M + 6) + 1$ bits. $M = \log_2 N$. 64: largest value of transform coefficients.
  - ✓ **1D Row Transform**: It computes $Y = ZD^T$. To map this operation to our 1D transform architecture, we compute $Y^T = DZ^T$, where Z is fed row-wise and Y^T is generated column-wise (or Y row-wise). So, functionally, our architecture maps exactly to the 1D row transform: it computes $Y = ZD^T$, where Z is fed row-wise and Y is generated row-wise.
    - Maximum output pixel value: $-2^{15} \times N \times 64 = -2^{15+M+6}$. The input sample is 16 bits wide, thus -2^{15} is the maximum absolute value. So, each pixel of the output Y requires $(15 + M + 6) + 1$ bits.
    - S_{T2}: HEVC requires each the pixels of Y to be pruned to 16 bits. Thus, we have to scale the outputs of the second 1D Transform so that they require at most 16 bits: $-2^{15+M+6} \times 2^x = -2^{15}$. Then $S_{T2} = 2^x = 2^{-(M+6)}$.
  - ✓ As Z must be fed row-wise, we need a transposition stage between the column and row transforms.

- This hardware core features the following parameters:
  - ✓ N: Size of transform (4, 8, 16, 32)
  - ✓ B: Bits per pixel of the input image. The input to the HEVC Transform is the residual image U with $B + 1$ bits per pixel.
  - ✓ FWD/INV: It indicates whether we want to compute the Forward of the Inverse HEVC Transform.
  - ✓ IMP: Implementation type:
    - Fully pipelined (2 transpose buffers): The 1D Transform Blocks are fully pipelined. Two transpose buffers (ping-pong mode) are used. The user can send one column per cycle within a block and between consecutive blocks.
    - One transpose buffer: The 1D Transform Blocks are fully pipelined. Only one transpose buffer is used. We can input one column per cycle within a block. However, to send the next block we have to wait $N - 1$ cycles.
    - Iterative (one transpose buffer): The 1D Transform Blocks are Iterative. Only one transpose buffer is used. Between columns, we have to wait certain number of cycles (as explained in a previous section). Between blocks, we also have to wait a certain number of cycles.



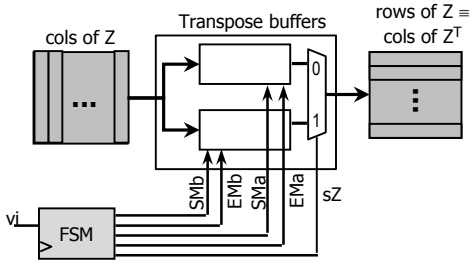
- **Transpose buffer** (or buffers):

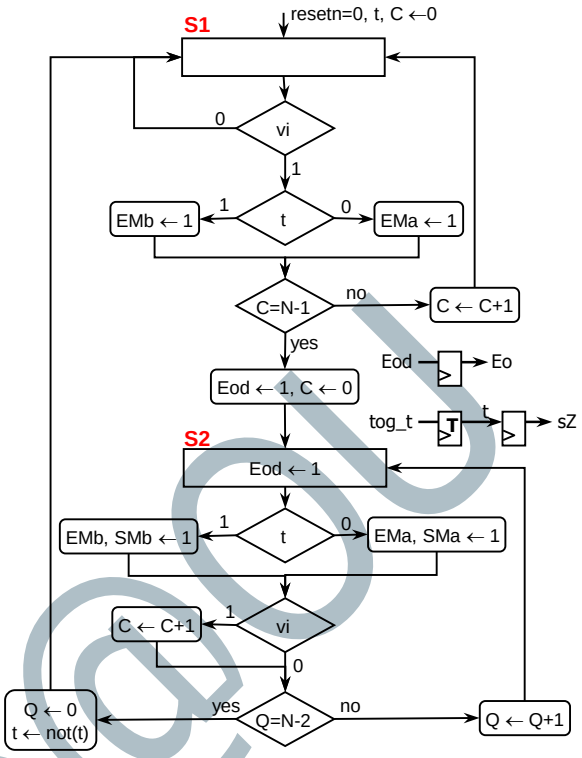


FULLY PIPELINED (TWO TRANSPOSE BUFFERS)

- Here, we use the fully pipelined 1D Transforms and two transpose buffers with a multiplexer.

- 1st Transform: We can feed a new column every clock cycle.
- 2nd Transform: We can feed a new column every clock cycle. This is automatically controlled by the FSM.
- Between input blocks, we can feed the next block on the next immediate cycle.
- We get one output row per cycle.





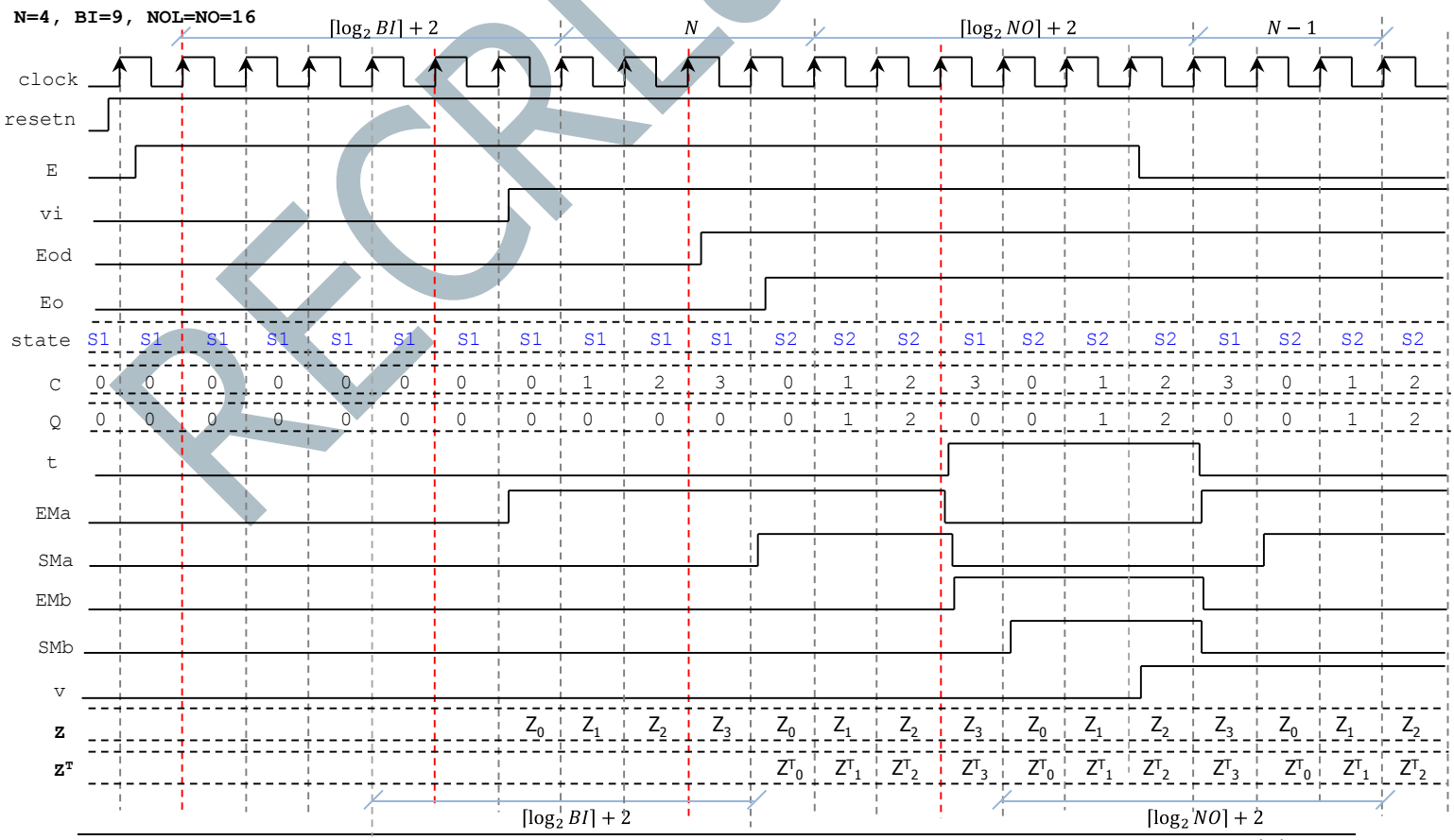
Execution time (for a NxN block):

- In the 1st Transform, for N columns, the time between first enable and last valid signal is $I/O\ delay_1 + N - 1$ (if we input a column per cycle). A cycle after that, we start the 2nd Transform, which always takes $I/O\ delay_2 + N - 1$. The total time (between first $E = 1$ and last $v = 1$) is:

$$I/O\ delay_1 + N + I/O\ delay_2 + N - 1,$$

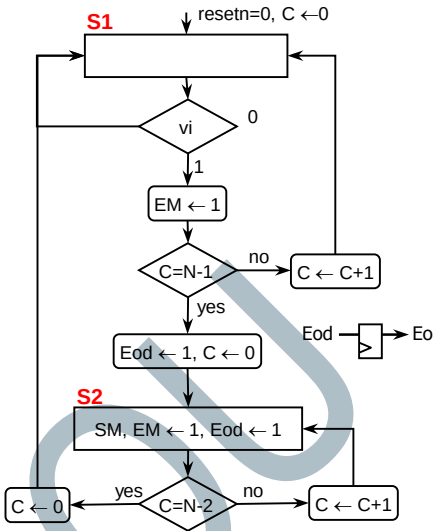
$$I/O\ delay_1 = \lceil \log_2(BI + \lceil \log_2(N/4) \rceil) \rceil + 2 + \lceil \log_2(N/4) \rceil,\ I/O\ delay_2 = \lceil \log_2(NO + \lceil \log_2(N/4) \rceil) \rceil + 2 + \lceil \log_2(N/4) \rceil$$

If we do not feed a column per cycle, the time between $E = 1$ and last $vi = 1$ is NOT $I/O\ delay_1 + N - 1$. In any case, there is no restriction at how fast we can input columns. In the figure, we use N=4, BI=9, thus the $I/O\ delay_1 = \lceil \log_2 BI \rceil + 2$:



ONE TRANSPOSE

- Here, we use the fully pipelined 1D Transforms and one transpose buffer. In this case, the FSM looks like this:

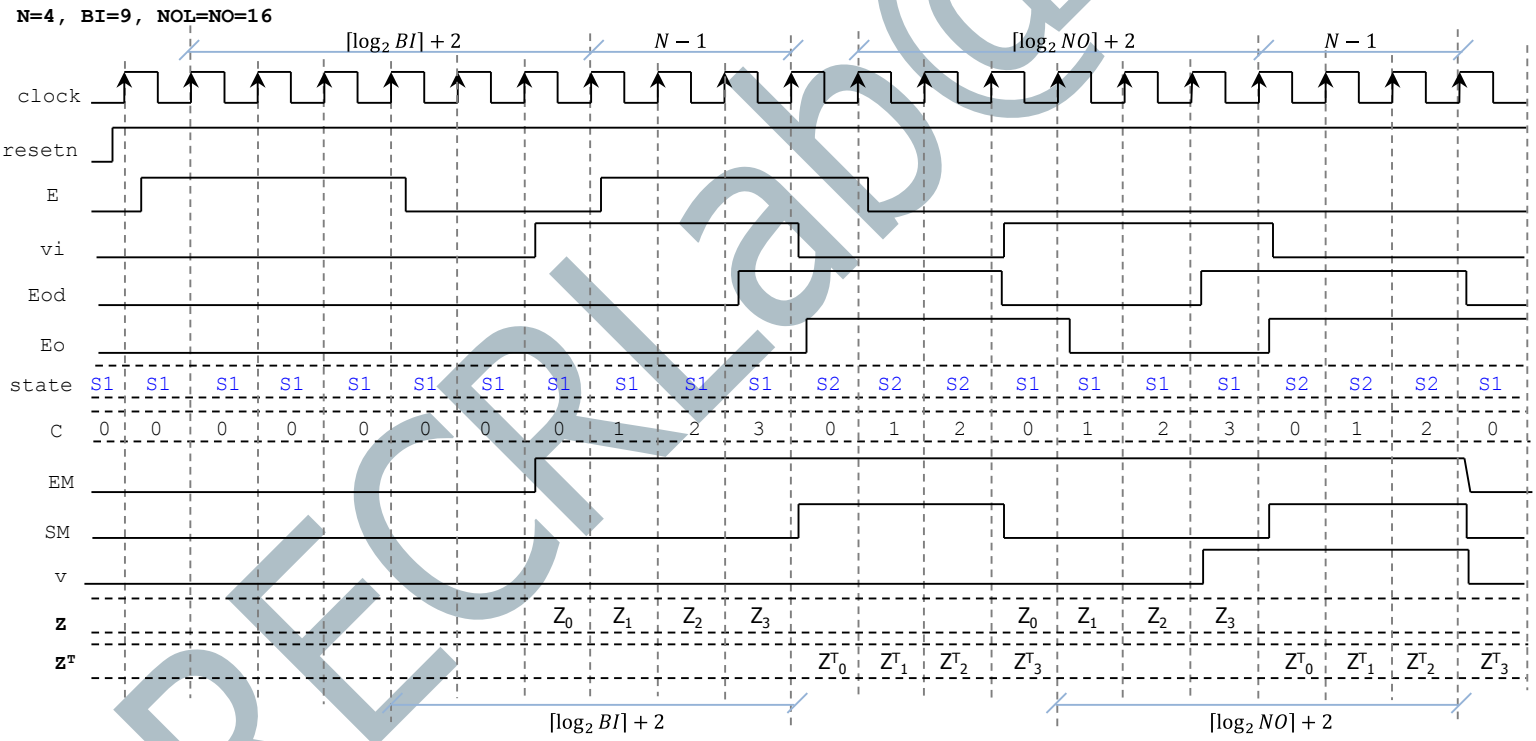


- 1st Transform: We can feed a new column every clock cycle.
- 2nd Transform: We can feed a new column every clock cycle. This is automatically controlled by the FSM.
- Between input blocks, we have to wait $N-1$ cycles before inputting a new column.
- Within an output block, we get one output row per cycle.
- **Execution time (for one block)**: Time between the first enable and the last valid signal from the 2nd Transform:

$$I/O\ delay_1 + N - 1 + 1 + I/O\ delay_2 + N - 1$$
$$I/O\ delay_1 = \lceil \log_2(BI + \lceil \log_2(N/4) \rceil) \rceil + 2 + \lceil \log_2(N/4) \rceil,$$
$$I/O\ delay_2 = \lceil \log_2(NO + \lceil \log_2(N/4) \rceil) \rceil + 2 + \lceil \log_2(N/4) \rceil$$

Of course, this time is valid assuming that we feed N input columns at every clock cycle. Within a block, we can feed one column per cycle.

In the figure, we use N=4, BI=9, thus $I/O\ delay_1 = \lceil \log_2 BI \rceil + 2$



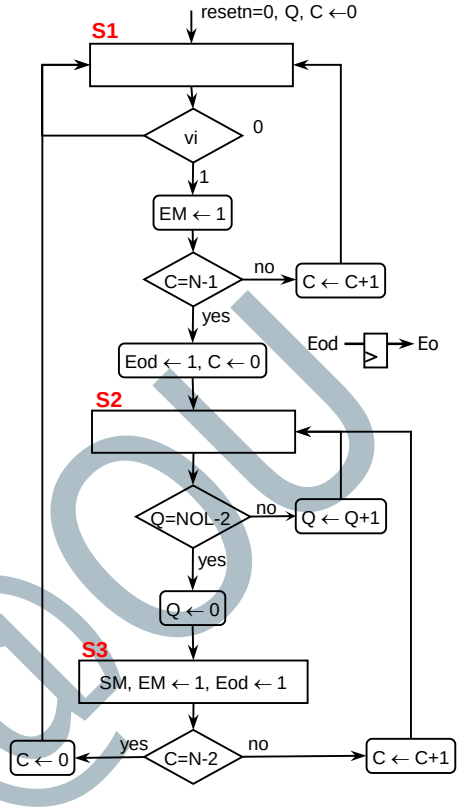
ITERATIVE

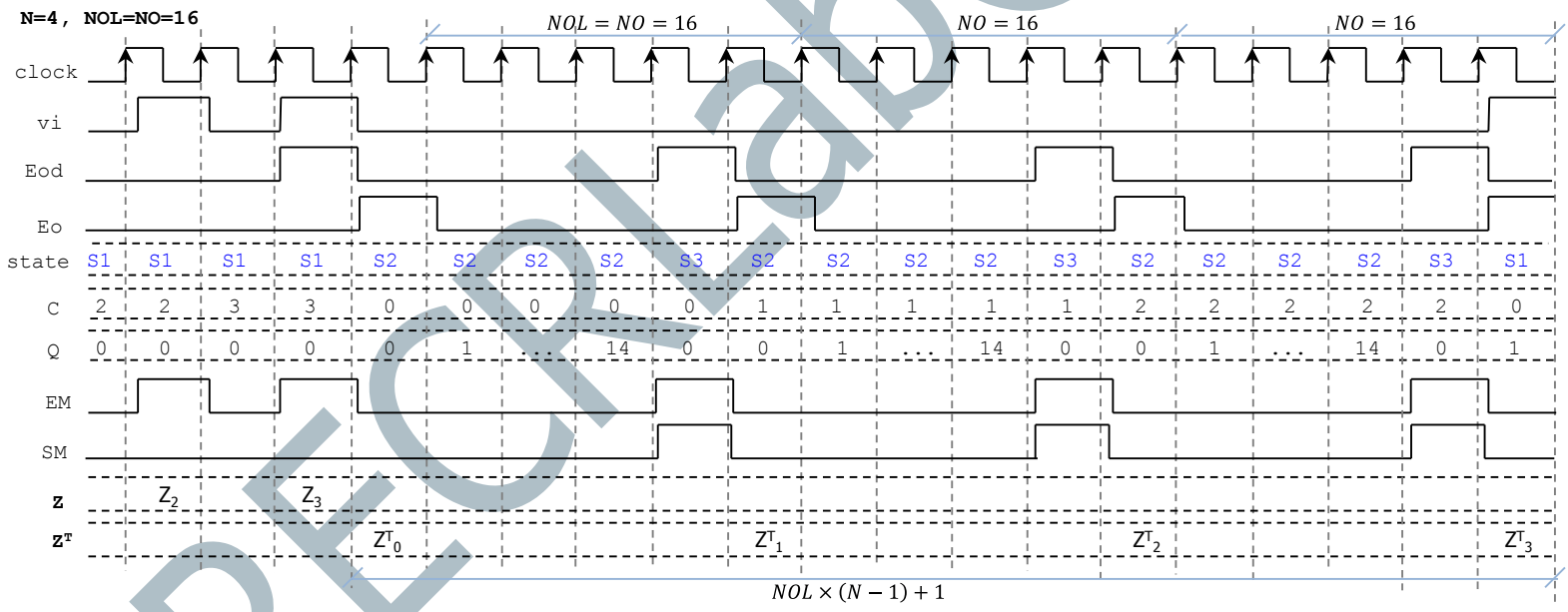
- Here, we use the iterative 1D Transforms and one transpose buffer. NO=16 for HEVC (we use NO here for the # of output bits of each 1D Transform)

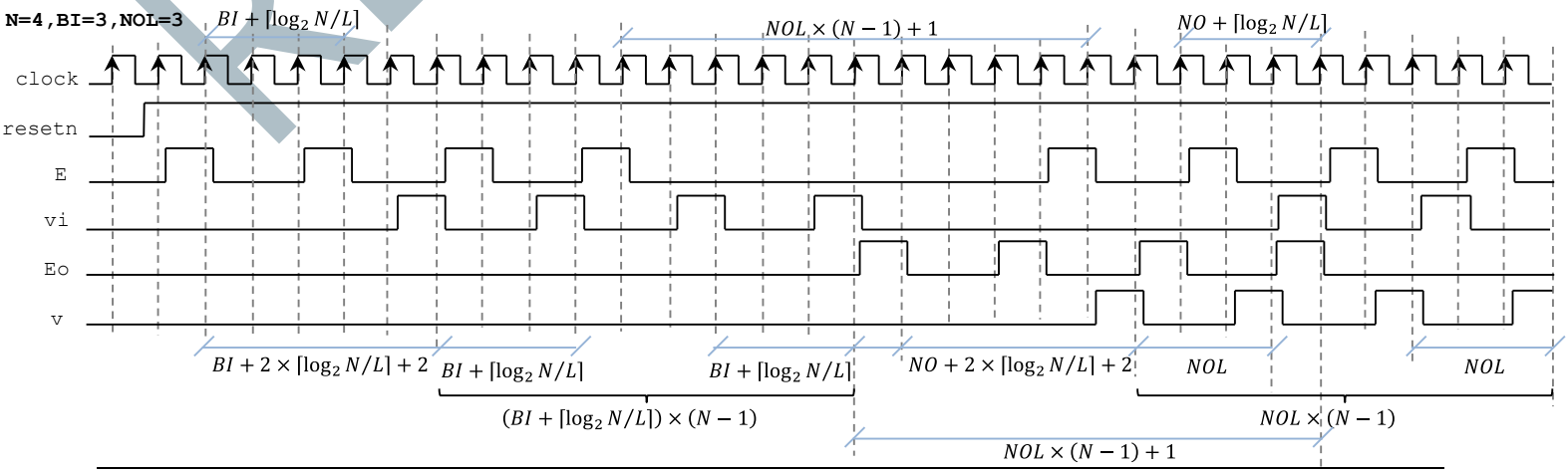
- 1st Transform: We have to wait $BIL = BI + \lceil \log_2(N/L) \rceil$ cycles between input columns. The I/O delay is $BI + 2 \times \lceil \log_2(N/L) \rceil + 2$.
- 2nd Transform: We have to wait $NOL = NO + \lceil \log_2(N/L) \rceil$ cycles between input columns (NO is the input bit-width here). This is automatically controlled by the FSM. The I/O delay is $NO + 2 \times \lceil \log_2(N/L) \rceil + 2$.
- Between input blocks, we have to wait $(NO + \lceil \log_2(N/L) \rceil) \times (N-1) + 1$ cycles before inputting a new column.

- **Execution time**: For an $N \times N$ input block, this is given by: time between first enable and last valid signal from 1st Transform, an extra cycle for the transpose buffer, and time between first enable and last valid signal from 2nd Transform:
$BIL(N-1) + BI + 2 \times \lceil \log_2(N/L) \rceil + 2 +$
$NOL(N-1) + 1 + NO + 2 \times \lceil \log_2(N/L) \rceil + 2 =$
$BI + NO + 4 \times \lceil \log_2(N/L) \rceil + 4 + (BIL + NOL)(N-1) + 1$

  For P consecutive NxN blocks we have:
$(BIL(N-1) + NOL(N-1) + 1)(P-1) + BIL(N-1) + BI + 2 \times \lceil \log_2(N/L) \rceil$
$+ 2 + NOL(N-1) + 1 + NO + 2 \times \lceil \log_2(N/L) \rceil + 2 =$
$(BIL(N-1) + NOL(N-1) + 1)P + BI + 2 \times \lceil \log_2(N/L) \rceil + 2 + NO +$
$2 \times \lceil \log_2(N/L) \rceil + 2$
  Of course, this time is valid assuming that we feed N input columns separated by BIL. If we do not do that and take longer, the execution time will change (the first part: time between first enable and last valid signal).





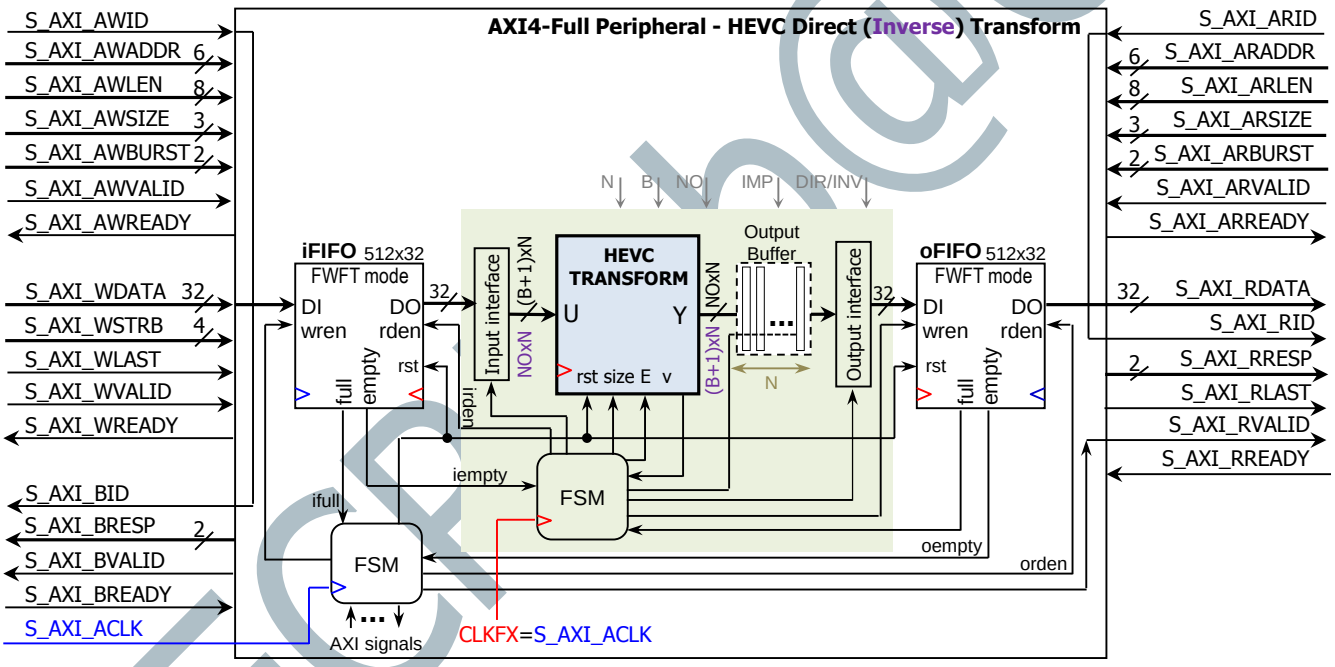


AXI4-FULL INTERFACING

- This is very similar to the PLB interface to the 2D DCT. Since the I/O data is 32-bits wide, we require an output buffer to hold data that usually appears at a faster rate that the oFIFO can retrieve.
- The HEVC transform specifies values for NO=16, and BI=B+1 (unlike the 2D DCT hardware). We use B=8.
- **Direct Transform**: $BI = B + 1$ bits for input. $NO = 16$ bits per output. Formula works for B=8, NO=16, AXIW=32. $NWIC = \frac{N}{AXIW/B}$, $NWOC = NWIC \times \left\lceil \frac{NO}{B} \right\rceil$.
- **Inverse Transform**: $BI = NO = 16$ bits per input. $NO = B + 1$ bits per output. Formula works for B=8, NO=16, AXIW=32 $NWIC = \frac{N}{AXIW/NO}$, $NWOC = \left\lceil \frac{N \times (B+1)}{AXIW} \right\rceil$
  NWIC: # of 32-bit words per input column. NWOC: # of 32-bit words per output row.

B=8, NO=16, AXIW=32		N=4	N=8	N=16	N=32
Direct	NWIC	1	2	4	8
	NWOC	2	4	8	16
Inverse	NWIC	2	4	8	16
	NWOC	2	3	5	9

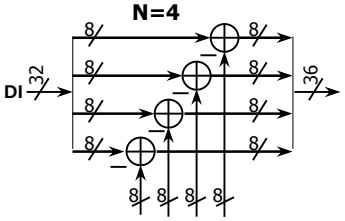
- There are two clock regions in the interface (for now we are setting S_AXI_ACLK=CLKFX), and one FSM for each clock region. The blue FSM is fixed and it manages the AXI interfacing. The red FSM manages the interface to the inner side of the FIFOs as well as the HEVC hardware and Output buffer.
- We require an output buffer to hold data that appears at a faster rate than what oFIFO can retrieve (this is because NO=16 and we get more than 32 bits per cycle on the output).

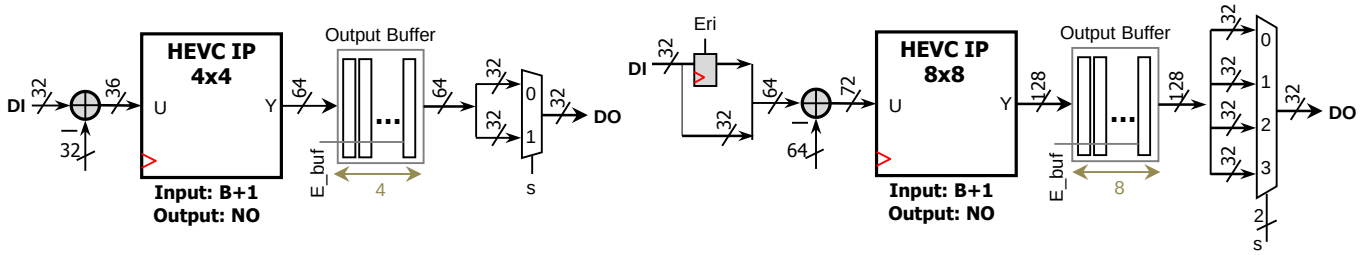


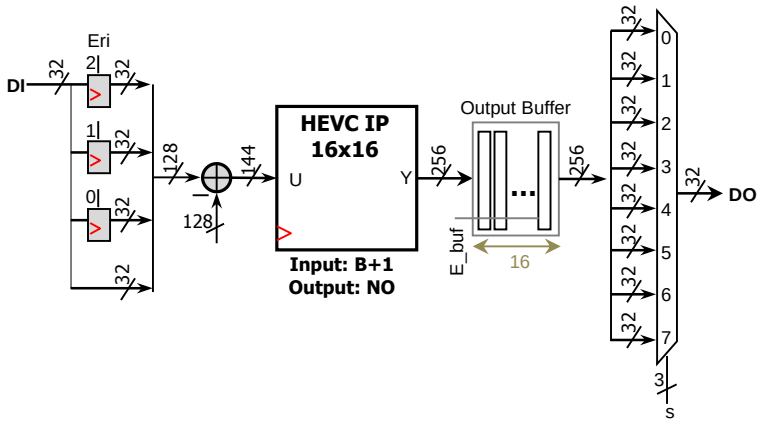
INPUT/OUTPUT INTERFACE

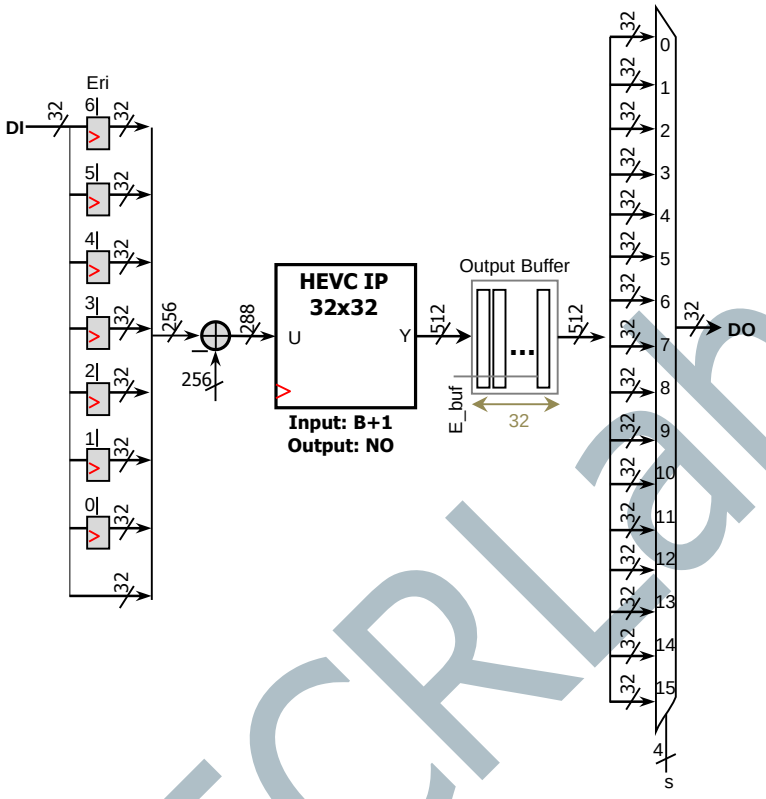
HEVC DIRECT TRANSFORM

- The figures below depict the different I/O interfaces. The iFIFO outputs 32 bit words and the oFIFO gets 32 bit words. We have to wait $NWIC$ cycles between input columns. Thus, if we have input/output data larger than 32 bits, we need some extra logic in the input and/or output.
- Note that bits per pixel is B bits. However we have BI=B+1 bits per pixels at the input of the HEVC Direct Transform. This is because the other image is coming from another hardware inside the FPGA (not from the processor). So we require N subtractors.



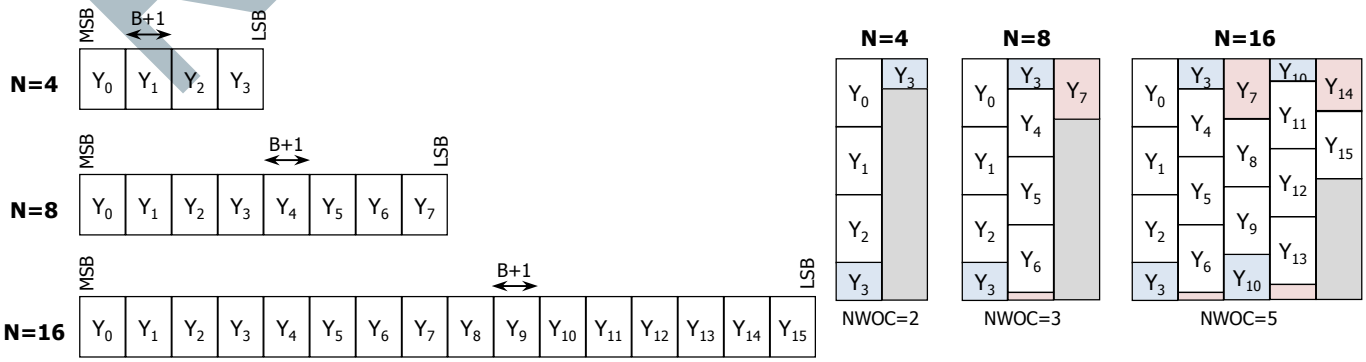


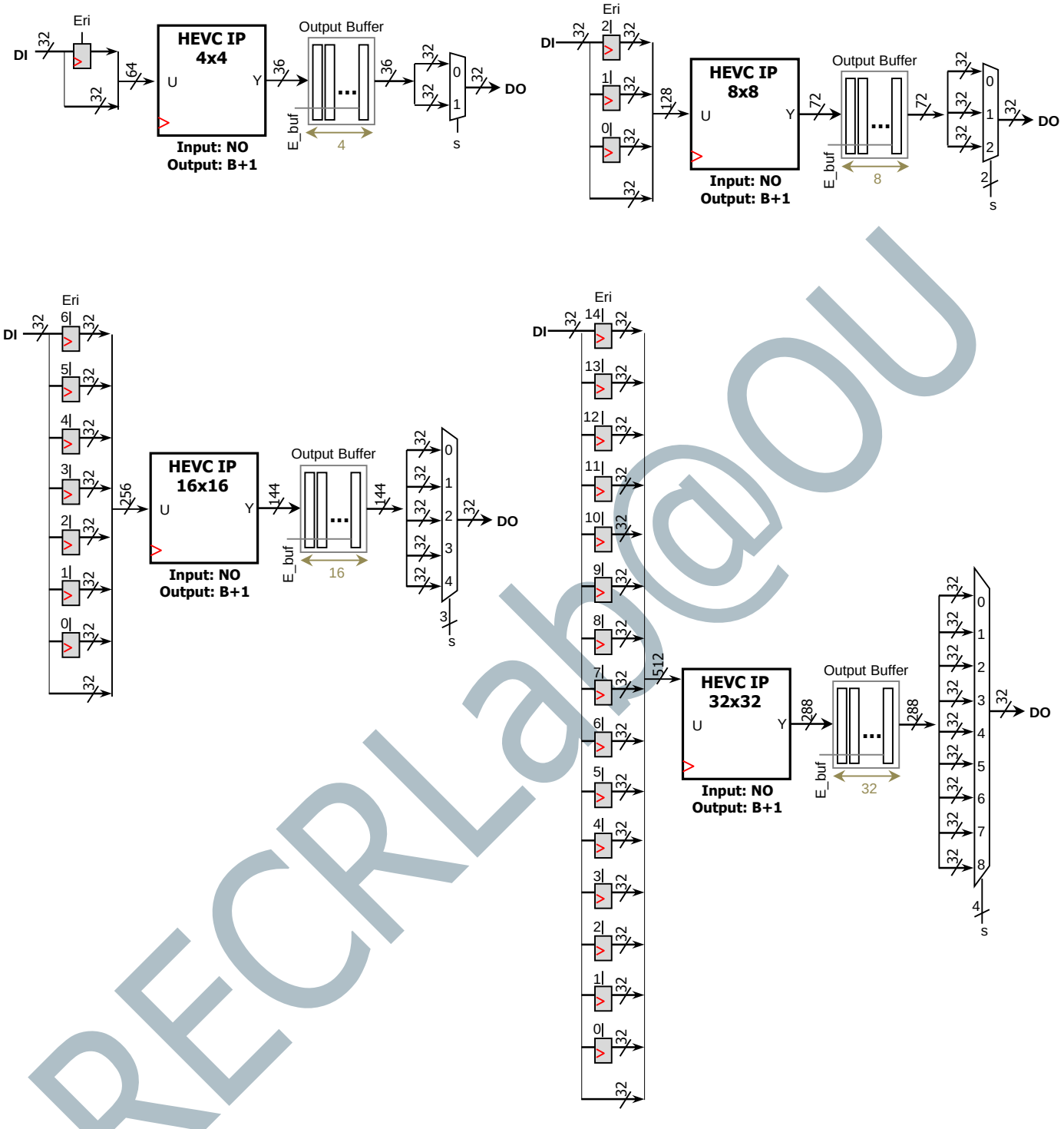




HEVC INVERSE TRANSFORM

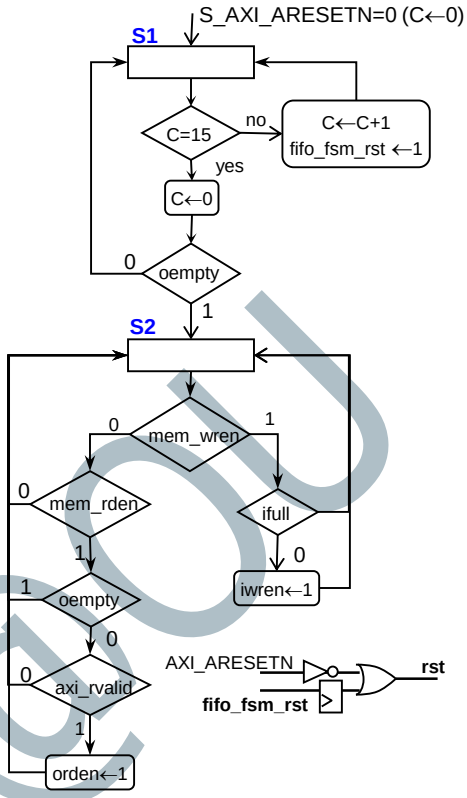
- The figures depict the different I/O interfaces. The iFIFO outputs 32 bit works and the oFIFO gets 32 bit words. Thus, whenever we have input/output data larger than 32 bits, we require some extra logic in the input and/or output.
- Output data arrangement: (for the input, it is just collection of 16-bit pixels, it is much easier). Here, we prefer to pack the entire output in a $(B+1) \times N$ array. Then we pad the array so it can be transmitted in multiples of 32 bits.




Eri
DI
32
HEVC IP
4x4
U
Y
Input: NO
Output: B+1
Output Buffer
E_buf
4
DO
HEVC IP
8x8
8
HEVC IP
16x16
16
HEVC IP
32x32
32

FSM @ S_AXI_ACLK (DOES NOT HANDLE PR_RESET DURING DPR)

- This blue FSM and the FIFOs (the green shaded circuit) do not change if we are using an AXI4-Full Interface. The red FSM does change to accommodate different circuits. Thus, we only need to interface to the FIFOs when dealing with different circuits.
- FIFOs have to be reset prior to usage for at least 3 read/write clock cycles. If we use 16 cycles @ 100 MHz, then the minimum clkfx is 16x10ns/3 = 53.33 ns → 18.75 MHz.
- The register to 'fifo_fsm_rst' is just to avoid glitches (not a big deal, it was done to avoid simulation problem as a reset to a FIFO has to be clean).
- This FSM does not change when the IP core inside the FIFOs change. This FSM is exactly the same for any AXI4-Full peripheral.
- The blue FSM changes depending on the type of interface (PLB, AXI). Once the blue FSM is designed (like in the figure) for the AXI interface, we reuse this circuit: now we only need to worry about designing a peripheral and an FSM in the CLKFX clock region.



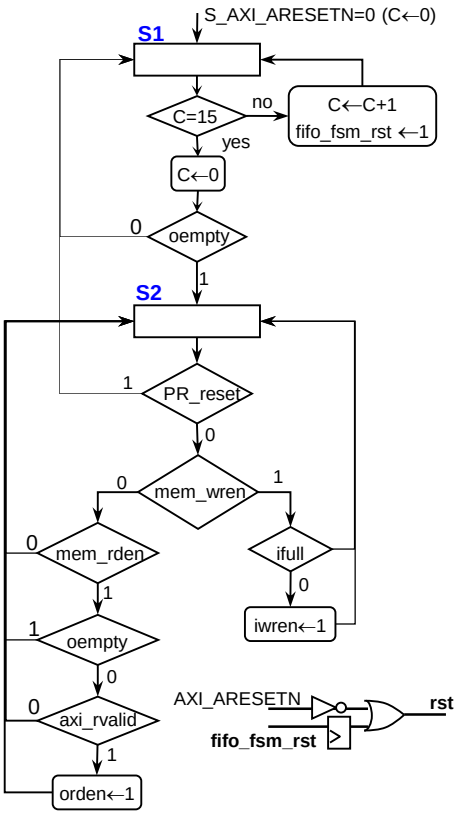
FSM at S_AXI_ACLK

FSM @ S_AXI_ACLK (IT HANDLES PR_RESET DURING DPR)

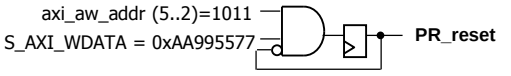
- This blue FSM and the FIFOs (the green shaded circuit) do not change when using an AXI4-Full Interface. The red FSM does change to accommodate different circuits. Thus, we only need to interface to the FIFOs when dealing with different circuits.
- FIFOs have to be reset prior to usage for at least 3 read/write clock cycles. If we use 16 cycles @ 100 MHz, then $clkfx_{min}$ = 16x10ns/3 = 53.33 ns → 18.75 MHz. Also $owren$ can only be used 2 cycles after reset.
- The register for 'fifo_fsm_rst' is just to avoid glitches (not a big deal, it was included to avoid simulation issues: a reset to a FIFO has to be clean).
- This FSM does not change when the IP core inside the FIFOs change. This FSM is exactly the same for any AXI4-Full peripheral.
- The blue FSM changes depending on the type of interface (PLB, AXI). Once the blue FSM is designed (as in the figure) for the AXI interface, we reuse this circuit: now we only need to worry about designing a peripheral and an FSM in the CLKFX clock region.

- Note that we included a new input: PR_reset. This allows us to reset the FIFO and the PR with a simple software command (we write onto address 101100 the word `0xAA995577`).
- PR_reset: This is the output of a flip flop. This signal is a pulse of one clock cycle. Every time axi_awaddr (latched S_AXI_AWADDR) and S_AXI_WDATA match what we want, we generate a pulse.

- Notice that once S_AXI_WDATA is captured by AXI, the latched address axi_awaddr increases its address by 4 (or changes). So we are sure that because this happens, PR_reset is only one pulse (and not a sequence of pulses created when axi_aw_addr keeps being the same)



FSM at S_AXI_ACLK

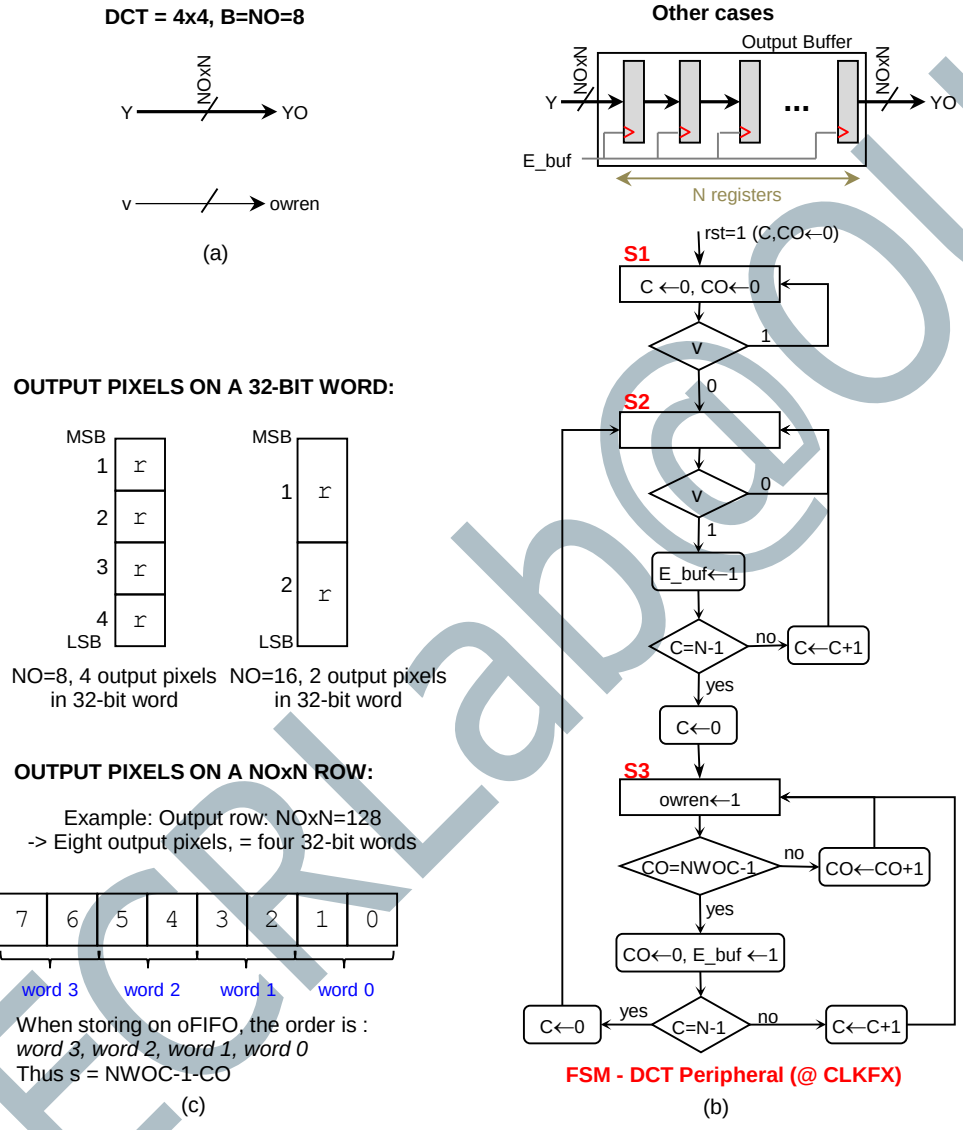


FSM @ CLKFX (HEVC DIRECT TRANSFORM)

- This consists of an input FSM and an output FSM. The generation of the signal Eri (depending on NWIC) remains the same as in the case of the 2D DCT PLB Interface.

OUTPUT FSM

- The output FSM and output buffer is the same as the one used in the 2D DCT PLB interface. Here, output bits NO can be 16 (Direct Transform) or (B+1) (Inverse Transform).



INPUT FSM

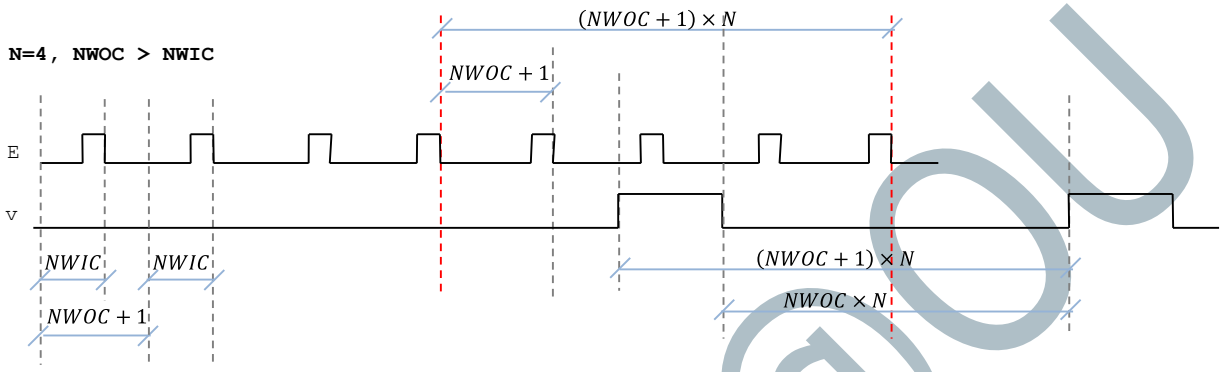
- There are three variations of the HEVC Transform core:
  - ✓ Fully Parallel (fullypip): 1D Transforms fully pipelined, two transpose memories.
  - ✓ One Transpose (onetras): 1D Transforms fully pipelined, one transpose memory.
  - ✓ Iterative, one Transpose (onetrans_iter): 1D Transforms iterative, one transpose memory.

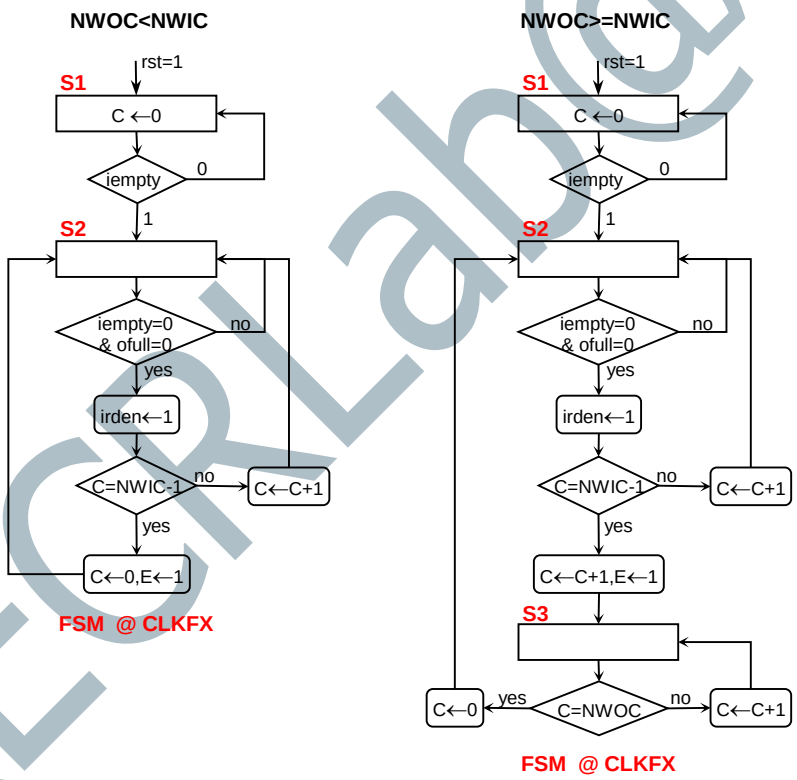
- HEVC Transform (Direct/Inverse): For the Direct case, the fullypip and onetrans case were updated (though the 2D DCT designs also worked). But, we designed from scratch the onetrans_iter (iterative) case. The generation of **Eri** is taken care of here. If NWIC=1, we do not need to generate Eri.

$$Eri = \frac{2^{NWIC-1-C}}{2(drop\ LSB)}\ AND\ irden$$

Fully Pipelined (fullypip):

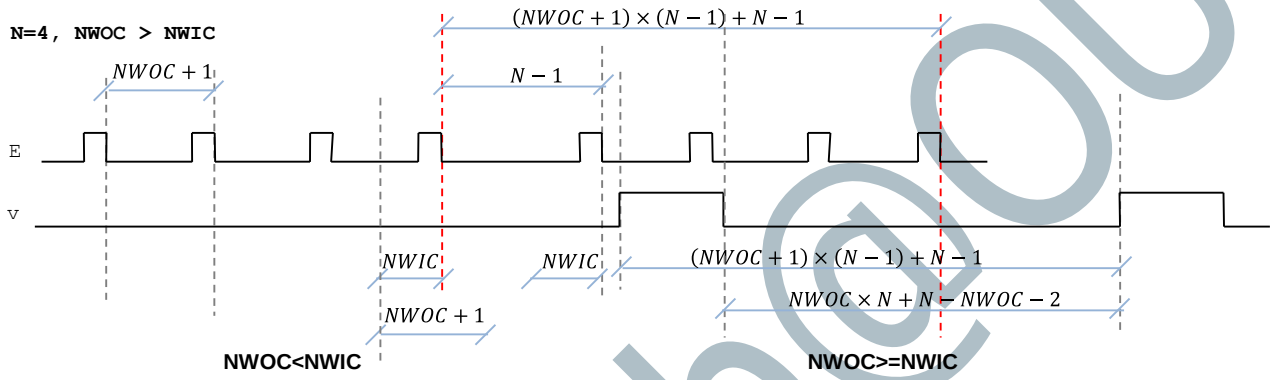
- Due to the input interface, we have to wait $NWIC$ cycles between input columns.
- Output FSM: Due to output interface, we must guarantee $NWOC \times N$ cycles between the last 'v' of a previous block and the first 'v' of the next block. This the same as waiting $NWOC \times N + N = (NWOC + 1) \times N$ cycles between the last E's of two consecutive blocks.
  - ✓ $NWOC \geq NWIC$: We can wait $NWOC + 1$ cycles between input columns. This way there are $(NWOC + 1) \times N$ cycles between the last columns of two consecutive blocks.
  - ✓ $NWOC < NWIC$: As we have to wait NWIC cycles between columns anyway, we wait $NWIC \times N$ cycles between the last columns of two consecutive blocks (this is more than $NWOC \times N$).



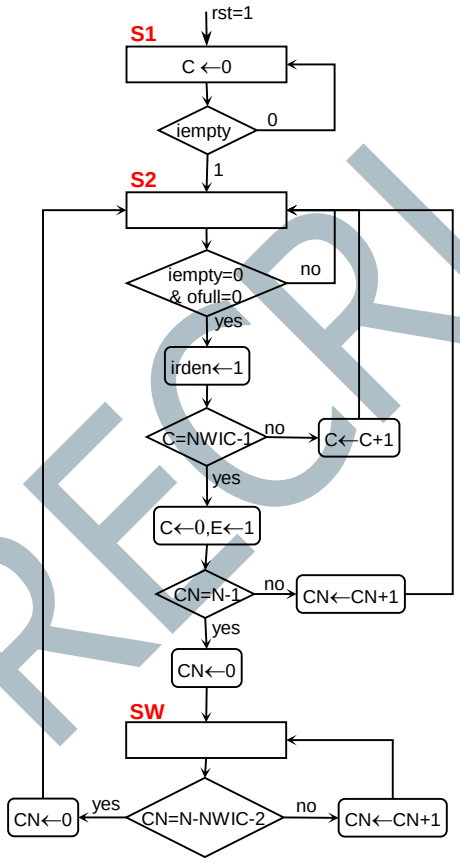


One Transpose (onetrans):

- Due to the input interface, we have to wait $NWIC$ cycles between input columns. Also, we have to have N-1 cycles between the last column of a previous block and the first column of the next block.
- The output FSM requires us to wait $NWOC \times N$ cycles between the last 'v' of a previous block and the first 'v' of the next block. This the same as waiting $(NWOC + 1) \times N$ cycles between the last E's of two consecutive blocks.
  - ✓ $NWOC \geq NWIC$: We wait $NWOC + 1$ cycles between input columns. We must wait $N - 1$ cycles between the last column of a block and the first column of next block (state SW: we wait $N - 1 - (NWOC + 1)$ cycles). So, between the last columns of two consecutive blocks, there are $(NWOC + 1) \times (N - 1) + N - 1 = NWOC \times N + N - NWOC - 2$ cycles. From S2 to S3, the FSM works for NWOC=NWIC, as we ask in S2 whether C=NWIC-1 but then ask in S3 if C=NWOC. If N=4, we never reach SW (we do not need it).
  - ✓ $NWOC < NWIC$: As we must wait $NWIC$ cycles between columns anyway, there are $NWIC \times N$ cycles between the last columns of two consecutive blocks (this is more than $NWOC \times N$). Between the last column of a block and the first column of next block we must wait $N - 1$ cycles (on state SW, we wait $N - 1 - NWIC$ cycles). This way we wait $NWIC \times (N - 1) + N - 1$ cycles between the last columns of two consecutive blocks.

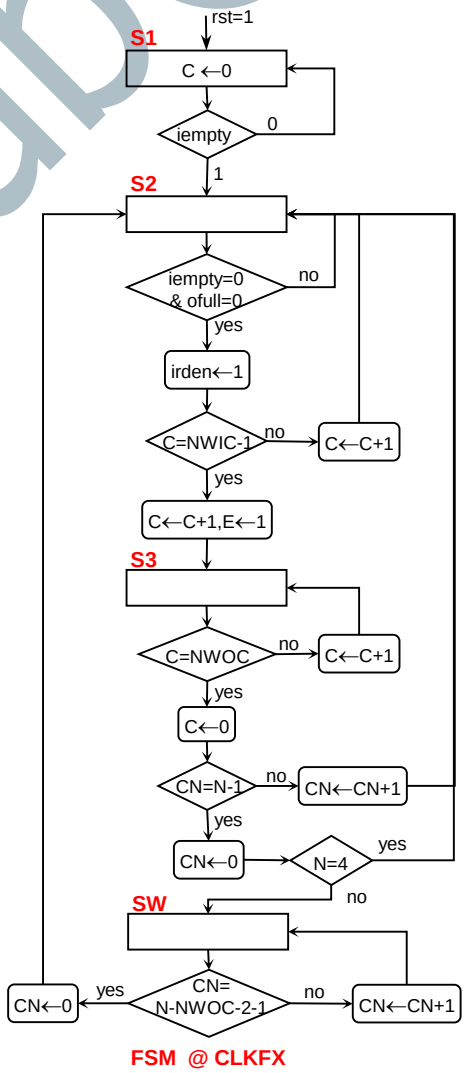


NWOC<NWIC



FSM @ CLKFX

NWOC>=NWIC



FSM @ CLKFX

Iterative, One Transpose (onetrans_iter)

- Direct Transform:
  - ✓ We wait BIL cycles between input columns. And between blocks, we wait $NOL \times (N-1)+1$ cycles. Here, $BI_D = B+1$, $NO_D = 16$. $BIL = BI_D + \log_2 N/L$, $NOL = NO_D + \log_2 N/L$.
  - ✓ In the FSM, after waiting $NWIC$ cycles to have an input column ready, we only wait $BIL - NWIC$ cycles before feeding the next column. Also, after waiting $NWIC$ cycles for the last input column to be ready, then we only wait $NOL \times (N-1)+1-NWIC$ cycles (the number of cycles is between the edges where 'E' is asserted) before feeding the next block.
  - ✓ The output FSM requires us to wait $NWOC \times N$ cycles between the last 'v' of a previous output block and the first 'v' of the new block (or $NWOC \times N + N$ between the last E signals of contiguous blocks). But for N=4,8,16,32, B=8, L=4, the following holds: $NWOC \times N \leq NOL \times (N-1)+1$. So, it's enough to wait $NOL \times (N-1)+1$ cycles.

- Inverse Transform:
  - ✓ We wait BI_I cycles between input columns. And between blocks, we wait $NO_I \times (N-1)+1$ cycles. Here, $BI_I = NO_D = 16$, $NO_I = BI_D = B+1$.
  - ✓ In the FSM, after waiting $NWIC$ cycles to have an input column ready, we only wait $BI_I - NWIC$ cycles before feeding the next column. Also, after waiting $NWIC$ cycles for the last input column to be ready, then we only wait $NO_I \times (N-1)+1-NWIC$ cycles (the number of cycles is between the edges where 'E' is asserted) before feeding the next block.
  - ✓ The output FSM requires us to wait $NWOC \times N$ cycles between the last 'v' of a previous output block and the first 'v' of the new block (or $NWOC \times N + N$ between the last E signals of contiguous blocks). For N=4,8,16, the following holds: $NWOC \times N \leq NO_I \times (N-1)+1$. For N=32, this doesn't hold. So, for simplicity's sake, we wait $NO_I \times (N-1)+1+NWIC \geq NWOC \times N$ cycles (N=4,8,16,32). After waiting $NWIC$ cycles to load the last column, we wait $NO_I \times (N-1)+1$ cycles. This is not the most efficient way (we are waiting extra cycles), but it is the simplest.