

Infringement Claim Chart for U.S. Patent No. 9,674,556 (the "'556 Patent")

As outlined in the below claim chart, Lenovo directly infringes, either literally or under the doctrine of equivalents, Claims 1-20 of the '556 Patent through their manufacture, sale for importation, importation, use, and sale within the United States after importation of one or more Accused Products that support the Alliance for Open Media ("AV1") video coding protocol.¹ Moreover, Lenovo contributes to the direct infringement by Lenovo Customers and/or actively induces Lenovo Customers to use the Accused Products such that the Lenovo Customers directly infringe at least the above asserted claims.

Appendix 1, served concurrently with this chart and which is incorporated by reference herein, identifies which of the Lenovo smartphones, desktop computers, laptop computers, and tablet computers are operable to decode according to the AV1 specification (the "Accused Products"). Appendix 1 includes a list of Lenovo products, including smartphones, desktop computers, laptop computers, and tablet computers, and lists the CPU and GPU chipsets contained within the Lenovo Products. Appendix 1 identifies public information, primarily published by the CPU and GPU chipset manufacturers, specifying whether the CPU and GPU chipsets are operable to decode according to the AV1 specification. For each Accused Product, Appendix 1 contains BATES number citations to such public information found and produced by InterDigital. Discovery in this case is at the outset, and discovery will further confirm that each Lenovo product is operable to decode AV1 video and thus infringe the asserted claims of the '556 Patent. The following chart is based on information known to date. Accordingly, InterDigital reserves the right to amend and update this chart as additional information is obtained and analyzed.² InterDigital further reserves the right to issue expert reports explaining in detail Lenovo's infringement of the '556 Patent.

¹ S1: Alliance for Open Media AV1 Version 1.0.0 with Errata 1 (2019-01-08 11:48 PT) as shown in IDG1373_0002997-3678, which is incorporated by reference herein in its entirety.

² This exemplary chart does not limit, and is not intended to limit, InterDigital's positions or contentions on claim construction, infringement, domestic industry, or validity. InterDigital reserves the right to rely on other claims and/or products to establish the technical prong of the domestic industry requirement or to establish infringement. Furthermore, this chart should not be construed as providing any positions on claim construction, such as, for example, whether any claim preamble should be limiting or whether any claim element is subject to 35 U.S.C. Section 112(f).

CLAIM	PROTOCOL
[1.P] A method comprising:	To the extent the preamble is limiting, the Accused Products either literally and/or under the doctrine of equivalents practice a method via the AV1 Protocol, as described by at least the following technical specification(s): S1: Alliance for Open Media AV1 Version 1.0.0 with Errata 1 (2019-01-08 11:48 PT) The products identified in Appendix 1 herewith, comprise a processor configured to decode video pursuant to the AV1 video coding protocol. <i>See also</i> IDG1373_0002182–2190 (Lenovo Website - Lenovo ThinkPad X1 Carbon Gen. 10); IDG1373_0003779–3795 (Intel website – 12th Gen Intel Mobile Product Brief, at 5, 8); IDG1373_0003718–3728 (Intel Website – Intel Processor Supported Codecs, at 6-7).
[1.1] decoding an image region,	The Accused Products either literally and/or under the doctrine of equivalents practice the element “decoding an image region” via the AV1 Protocol, as described by at least the following technical specification(s). For example, the Accused Products identified in Appendix 1 practice the AV1 Specification, and, as shown below, the AV1 Specification specifies that the image region is decoded. Therefore, the Accused Products, through operability with the AV1 Specification, practice Claim Element 1.1. S1: § 1. Scope This document specifies the Alliance for Open Media AV1 bitstream formats and decoding process. <u>The Alliance for Open Media AV1 bitstream utilizes the below process for decoding an image region.</u> S1: § 7. Decoding process S1: § 7.1. Overview AV1 contains two operating modes: 1. General decoding (input is a sequence of OBUs, output is decoded frames) 2. Large scale tile decoding (input is a tile list OBU plus additional side information, output is a decoded frame) The general decoding process is specified in section 7.2. S1: § 7.2 General decoding process

CLAIM	PROTOCOL																		
	When film_grain_params_present is equal to 0, decoders shall produce output frames that are identical in all respects and have the same output order as those produced by the decoding process specified herein. (...) The input to this process is a sequence of open bitstream units (OBUs). The output from this process is a sequence of decoded frames.																		
[1.2] wherein the decoding includes performing in-loop filtering to reduce at least a first and a second type of coding artifact using at least two filters in succession,	The Accused Products either literally and/or under the doctrine of equivalents practice the element “wherein the decoding includes performing in-loop filtering to reduce at least a first and a second type of coding artifact using at least two filters in succession,” via the AV1 Protocol, as described by at least the following technical specification(s). For example, the Accused Products identified in Appendix 1 practice the AV1 Specification, and, as shown below, the AV1 Specification specifies that the in-loop filtering uses at least two filters in succession, a loop filter and a CDEF filter, to reduce coding artifacts when decoding the image region. Therefore, the Accused Products, through operability with the AV1 Specification, practice Claim Element 1.2. <u>In the AV1 specification, the frame is coded in the Frame OBU syntax table which includes a frame header called from frame_header_obu() and the coded frame data coded in tile_group_obu (sz) syntax table.</u> S1: § 5.10. Frame OBU syntax <table border="1" data-bbox="464 1036 1898 1369"> <thead> <tr> <th>frame_obu(sz) {</th><th>Type</th></tr> </thead> <tbody> <tr> <td>startBitPos = get_position()</td><td></td></tr> <tr> <td>frame_header_obu()</td><td></td></tr> <tr> <td>byte_alignment()</td><td></td></tr> <tr> <td>endBitPos = get_position()</td><td></td></tr> <tr> <td>headerBytes = (endBitPos - startBitPos) / 8</td><td></td></tr> <tr> <td>sz -= headerBytes</td><td></td></tr> <tr> <td>tile_group_obu(sz)</td><td></td></tr> <tr> <td>}</td><td></td></tr> </tbody> </table>	frame_obu(sz) {	Type	startBitPos = get_position()		frame_header_obu()		byte_alignment()		endBitPos = get_position()		headerBytes = (endBitPos - startBitPos) / 8		sz -= headerBytes		tile_group_obu(sz)		}	
frame_obu(sz) {	Type																		
startBitPos = get_position()																			
frame_header_obu()																			
byte_alignment()																			
endBitPos = get_position()																			
headerBytes = (endBitPos - startBitPos) / 8																			
sz -= headerBytes																			
tile_group_obu(sz)																			
}																			

CLAIM	PROTOCOL																														
	S1: § 6.9. Frame OBU semantics A frame OBU consists of a frame header OBU and a tile group OBU packed into a single OBU. Note: The intention is to provide a more compact way of coding the common use case where the frame header is immediately followed by tile group data. S1: § 5.11. Tile group OBU syntax S1: § 5.11.P. General tile group OBU syntax <table border="1"> <thead> <tr> <th>tile_group_obu(sz) {</th><th>Type</th></tr> </thead> <tbody> <tr> <td>NumTiles = TileCols * TileRows</td><td></td></tr> <tr> <td>startBitPos = get_position()</td><td></td></tr> <tr> <td>tile_start_and_end_present_flag = 0</td><td></td></tr> <tr> <td>if (NumTiles > 1)</td><td></td></tr> <tr> <td>tile_start_and_end_present_flag</td><td>f(1)</td></tr> <tr> <td>if (NumTiles == 1 !tile_start_and_end_present_flag) {</td><td></td></tr> <tr> <td>tg_start = 0</td><td></td></tr> <tr> <td>tg_end = NumTiles - 1</td><td></td></tr> <tr> <td>} else {</td><td></td></tr> <tr> <td>tileBits = TileColsLog2 + TileRowsLog2</td><td></td></tr> <tr> <td>tg_start</td><td>f(tileBits)</td></tr> <tr> <td>tg_end</td><td>f(tileBits)</td></tr> <tr> <td>}</td><td></td></tr> <tr> <td>byte_alignment()</td><td></td></tr> </tbody> </table>	tile_group_obu(sz) {	Type	NumTiles = TileCols * TileRows		startBitPos = get_position()		tile_start_and_end_present_flag = 0		if (NumTiles > 1)		tile_start_and_end_present_flag	f(1)	if (NumTiles == 1 !tile_start_and_end_present_flag) {		tg_start = 0		tg_end = NumTiles - 1		} else {		tileBits = TileColsLog2 + TileRowsLog2		tg_start	f(tileBits)	tg_end	f(tileBits)	}		byte_alignment()	
tile_group_obu(sz) {	Type																														
NumTiles = TileCols * TileRows																															
startBitPos = get_position()																															
tile_start_and_end_present_flag = 0																															
if (NumTiles > 1)																															
tile_start_and_end_present_flag	f(1)																														
if (NumTiles == 1 !tile_start_and_end_present_flag) {																															
tg_start = 0																															
tg_end = NumTiles - 1																															
} else {																															
tileBits = TileColsLog2 + TileRowsLog2																															
tg_start	f(tileBits)																														
tg_end	f(tileBits)																														
}																															
byte_alignment()																															

endBitPos = get_position()	
headerBytes = (endBitPos - startBitPos) / 8	
sz -= headerBytes	
for (TileNum = tg_start; TileNum <= tg_end; TileNum++) {	
tileRow = TileNum / TileCols	
tileCol = TileNum % TileCols	
lastTile = TileNum == tg_end	
if (lastTile) {	
tileSize = sz	
} else {	
tile_size_minus_1	le(TileSizeBytes)
tileSize = tile_size_minus_1 + 1	
sz -= tileSize + TileSizeBytes	
}	
MiRowStart = MiRowStarts[tileRow]	
MiRowEnd = MiRowStarts[tileRow + 1]	
MiColStart = MiColStarts[tileCol]	
MiColEnd = MiColStarts[tileCol + 1]	
CurrentQIndex = base_q_idx	
init_symbol(tileSize)	
decode_tile()	
exit_symbol()	
}	
if (tg_end == NumTiles - 1) {	
if (!disable_frame_end_update_cdf) {	
frame_end_update_cdf()	
}	
decode_frame_wrapup()	
SeenFrameHeader = 0	
}	
}	

CLAIM	PROTOCOL
	<u>After the decoding of the tiles of the frame invoked from decode_tile (), the function decode_frame_wrapup () is invoked. The decode_frame_wrapup function of AV1 is described in § 6.10.1, below:</u> S1: § 6.10. Tile group OBU semantics S1: § 6.10.1. General tile group OBU semantics decode_frame_wrapup is a function call that indicates that the decode frame wrapup process specified in section 7.4 should be invoked. <u>The decode_frame_wrapup process to perform in-loop filtering to reduce at least a first and a second type of coding artifact using at least two filters in succession results from the decode_frame_wrapup function call, and is described in § 7.4, below:</u> S1: § 7.4. Decode frame wrapup process This process is triggered by a call to decode_frame_wrapup from within the syntax tables. At this stage, all the tile level decode has been done, and this process performs any frame level decode that is required. If show_existing_frame is equal to 0, the process first performs any post processing filtering by the following ordered steps: 1. If loop_filter_level[0] is not equal to 0 or loop_filter_level[1] is not equal to 0, the loop filter process specified in section 7.14 is invoked (this process modifies the contents of CurrFrame). 2. The CDEF process specified in section 7.15 is invoked (this process takes CurrFrame and produces CdefFrame). <u>AV1's loop filter process, the first of the at least two filters in succession, is specified in § 7.14, below:</u> S1: § 7.14. Loop filter process S1: § 7.14.1 General Input to this process is the array CurrFrame of reconstructed samples.

CLAIM	PROTOCOL
	Output from this process is a modified array CurrFrame containing deblocked samples. The purpose of the loop filter is to eliminate (or at least reduce) visually objectionable artifacts associated with the semi- independence of the coding of super blocks and their constituent sub-blocks. The loop filter is applied on all vertical boundaries followed by all horizontal boundaries as follows: <pre> for (plane = 0; plane < NumPlanes; plane++) { if (plane == 0 loop_filter_level[1 + plane]) { for (pass = 0; pass < 2; pass++) { rowStep = (plane == 0) ? 1 : (1 << subsampling_y) colStep = (plane == 0) ? 1 : (1 << subsampling_x) for (row = 0; row < MiRows; row += rowStep) for (col = 0; col < MiCols; col += colStep) loop_filter_edge(plane, pass, row, col) } } } </pre> <u>AV1's CDEF process, the second of the at least two filters in succession, is specified in § 7.15, below:</u> S1: § 7.15. CDEF process Input to this process is the array CurrFrame of reconstructed samples. Output from this process is the array CdefFrame containing deringed samples. The purpose of CDEF is to perform deringing based on the detected direction of blocks.

CLAIM	PROTOCOL
[1.3] the at least two filters including a deblocking filter for performing a first pass to reduce blocking artifacts	The Accused Products either literally and/or under the doctrine of equivalents practice the element “the at least two filters including a deblocking filter for performing a first pass to reduce blocking artifacts” via the AV1 Protocol, as described by at least the following technical specification(s). For example, the Accused Products identified in Appendix 1 practice the AV1 Specification, and, as shown below, the AV1 Specification specifies that a loop filter process is used to perform a first pass to reduce blocking artifacts and the output of this process is deblocked samples. Therefore, the loop filter described in the AV1 Specification is a deblocking filter for performing a first pass to reduce blocking artifacts. Accordingly, the Accused Products, through operability with the AV1 Specification, practice Claim Element 1.3. <u>As described with respect to Claim Element 1.2, the decode_frame_wrapup syntax included in the AV1 General tile group OBU syntax is used to call the decode_frame_wrapup function, which triggers the decode frame wrapup process, described in § 7.4, below:</u> S1: § 7.4. Decode frame wrapup process This process is triggered by a call to decode_frame_wrapup from within the syntax tables. At this stage, all the tile level decode has been done, and this process performs any frame level decode that is required. If show_existing_frame is equal to 0, the process first performs any post processing filtering by the following ordered steps: 1. If loop_filter_level[0] is not equal to 0 or loop_filter_level[1] is not equal to 0, the loop filter process specified in section 7.14 is invoked (this process modifies the contents of CurrFrame). 2. The CDEF process specified in section 7.15 is invoked (this process takes CurrFrame and produces CdefFrame). <u>The decode frame wrapup process performs filtering by invoking the ordered steps of (1) the loop filter process and (2) the CDEF process. Accordingly, AV1's loop filter process is the deblocking filter for performing a first pass to reduce blocking artifacts, as is specified in § 7.14, below:</u> S1: § 7.14. Loop filter process S1: § 7.14.1 General Input to this process is the array CurrFrame of reconstructed samples. Output from this process is a modified array CurrFrame containing deblocked samples.

CLAIM	PROTOCOL
	The purpose of the loop filter is to eliminate (or at least reduce) visually objectionable artifacts associated with the semi- independence of the coding of super blocks and their constituent sub-blocks. The loop filter is applied on all vertical boundaries followed by all horizontal boundaries as follows: <pre> for (plane = 0; plane < NumPlanes; plane++) { if (plane == 0 loop_filter_level[1 + plane]) { for (pass = 0; pass < 2; pass++) { rowStep = (plane == 0) ? 1 : (1 << subsampling_y) colStep = (plane == 0) ? 1 : (1 << subsampling_x) for (row = 0; row < MiRows; row += rowStep) for (col = 0; col < MiCols; col += colStep) loop_filter_edge(plane, pass, row, col) } } } </pre> <u>AV1's loop filter is an integral part of the decoding process:</u> Note: The loop filter is an integral part of the decoding process, in that the results of loop filtering are used in the prediction of subsequent frames.
[1.4] and an adaptive sparse de-noising filter for performing a second pass to reduce noise,	The Accused Products either literally and/or under the doctrine of equivalents practice the element “an adaptive sparse de-noising filter for performing a second pass to reduce noise” via the AV1 Protocol, as described by at least the following technical specification(s). For example, the Accused Products identified in Appendix 1 practice the AV1 Specification, and, as shown below, the AV1 Specification specifies that a CDEF process is used to perform a second pass to reduce noise and the output of this process is deringed samples. Therefore, the CDEF filter described in the AV1 Specification is an adaptive sparse denoising filter for performing a second pass to reduce noise. Accordingly, the Accused Products, through operability with the AV1 Specification, practice Claim Element 1.4.

CLAIM	PROTOCOL
	<u>The decode frame wrapup process performs filtering by invoking the ordered steps of (1) the loop filter process and (2) the CDEF process. The AV1 CDEF uses a non-linear low-pass filter designed to remove coding artifacts without blurring sharp edges. See, e.g., Midtskogen, et. al, "The AV1 Constrained Directional Enhancement Filter (CDEF)," April 2018, Conference: ICASSP 2018 - 2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), at 3. The filter works by identifying the direction of each block and then adaptively filtering with a high degree of control over the filter strength along the direction and across it. <i>Id.</i>, at 1. The CDEF filter adapts to the sparse nature of underlying video signals by "preventing excessive blurring when the filter is applied across an edge." <i>Id.</i>, at 3. This is done by using the "constraint function $f(d, S, D)$ operating on the difference between the filtered pixel and each of the neighboring pixels. For small differences, $f(d, S, D) = d$, making the filter in (6) behave like a linear filter. When the difference is large, $f(d, S, D) = 0$, which effectively ignores the filter taps" and leads to no filtering. <i>Id.</i> at 3. Accordingly, AV1's CDEF filter process is the adaptive sparse denoising filter for performing a second pass to reduce noise, as is specified in § 7.15, below:</u> S1: § 7.15. CDEF process Input to this process is the array CurrFrame of reconstructed samples. Output from this process is the array CdefFrame containing 10eranged samples. The purpose of CDEF is to perform deringing based on the detected direction of blocks. CDEF parameters are stored for each 64 by 64 block of luma samples. The CDEF filter is applied on each 8 by 8 block as follows:

CLAIM	PROTOCOL
	<pre data-bbox="485 289 1923 737"> step4 = Num_4x4_Blocks_Wide[BLOCK_8X8] cdefSize4 = Num_4x4_Blocks_Wide[BLOCK_64X64] cdefMask4 = ~(cdefSize4 - 1) for (r = 0; r < MiRows; r += step4) { for (c = 0; c < MiCols; c += step4) { baseR = r & cdefMask4 baseC = c & cdefMask4 idx = cdef_idx[baseR][baseC] cdef_block(r, c, idx) } } </pre> When the <code>cdef_block</code> function is called, the CDEF block process specified in section 7.15.1 is invoked with <code>r</code>, <code>c</code>, and <code>idx</code> as inputs. <u>The CDEF block process, which in turn invokes the CDEF filter process with the inputs 0, <code>r</code>, <code>c</code>, <code>priStr</code>, <code>secStr</code>, <code>damping</code>, and <code>dir</code>, results from the <code>cdef_block</code> function call, and is described in § 7.15.1, below:</u> S1: § 7.15.1. CDEF block process The inputs to this process are: • variables <code>r</code> and <code>c</code> specifying the location of an 8x8 block in units of 4x4 blocks in the luma plane, • a variable <code>idx</code> specifying which set of CDEF parameters to use, or -1 to signal that no filtering should be applied. The block is first copied to the <code>CdefFrame</code> as follows:

CLAIM	PROTOCOL
	<pre> startY = r * MI_SIZE endY = startY + MI_SIZE * 2 startX = c * MI_SIZE endX = startX + MI_SIZE * 2 for (y = startY; y < endY; y++) { for (x = startX; x < endX; x++) { CdefFrame[0][y][x] = CurrFrame[0][y][x] } } if (NumPlanes > 1) { startY >>= subsampling_y endY >>= subsampling_y startX >>= subsampling_x endX >>= subsampling_x for (y = startY; y < endY; y++) { for (x = startX; x < endX; x++) { CdefFrame[1][y][x] = CurrFrame[1][y][x] CdefFrame[2][y][x] = CurrFrame[2][y][x] } } } </pre> (...) The variable skip is set equal to (Skips[r][c] && Skips[r + 1][c] && Skips[r][c + 1] && Skips[r + 1][c + 1]). If skip is equal to 0, the CDEF direction process specified in section 7.15.2 is invoked with r and c as inputs, and the outputs assigned to variables yDir and var. If skip is equal to 0, the following ordered steps apply: 1. The variable priStr is set equal to cdef_y_pri_strength[idx] << coeffShift.

CLAIM	PROTOCOL
	2. The variable secStr is set equal to $\text{cdef_y_sec_strength}[\text{idx}] \ll \text{coeffShift}$. 3. The variable dir is set equal to $(\text{priStr} == 0) ? 0 : \text{yDir}$. 4. The variable varStr is set equal to $(\text{var} \gg 6) ? \text{Min}(\text{FloorLog2}(\text{var} \gg 6), 12) : 0$. 5. The variable priStr is set equal to $(\text{var} ? (\text{priStr} * (4 + \text{varStr}) + 8) \gg 4 : 0)$. 6. The variable damping is set equal to $\text{CdefDamping} + \text{coeffShift}$. 7. The CDEF filter process specified in section 7.15.3 is invoked with plane equal to 0, r, c, priStr, secStr, damping, and dir as input. 8. If NumPlanes is equal to 1, the process terminates at this point (i.e. filtering is not done for the U and V planes). 9. The variable priStr is set equal to $\text{cdef_uv_pri_strength}[\text{idx}] \ll \text{coeffShift}$. 10. The variable secStr is set equal to $\text{cdef_uv_sec_strength}[\text{idx}] \ll \text{coeffShift}$. 11. The variable dir is set equal to $(\text{priStr} == 0) ? 0 : \text{Cdef_Uv_Dir}[\text{subsampling_x}][\text{subsampling_y}][\text{yDir}]$. 12. The variable damping is set equal to $\text{CdefDamping} + \text{coeffShift} - 1$. 13. The CDEF filter process specified in section 7.15.3 is invoked with plane equal to 1, r, c, priStr, secStr, damping, and dir as input.

CLAIM	PROTOCOL
	14. The CDEF filter process specified in section 7.15.3 is invoked with plane equal to 2, r, c, priStr, secStr, damping, and dir as input. Cdef_Uv_Dir is a constant lookup table defined as: <pre data-bbox="464 423 1908 699"> Cdef_Uv_Dir[2][2][8] = { { {0, 1, 2, 3, 4, 5, 6, 7}, {1, 2, 2, 2, 3, 4, 6, 0} }, { {7, 0, 2, 4, 5, 6, 6, 6}, {0, 1, 2, 3, 4, 5, 6, 7} } }</pre> <u>The CDEF params syntax are described in § 5.9.19, below:</u> S1: § 5.9.19. CDEF params syntax

CLAIM	PROTOCOL																																																								
	<table> <tr> <th></th><th>Type</th></tr> <tr><td>cdef_params() {</td><td></td></tr> <tr><td> if (CodedLossless allow_intrabc </td><td></td></tr> <tr><td> !enable_cdef) {</td><td></td></tr> <tr><td> cdef_bits = 0</td><td></td></tr> <tr><td> cdef_y_pri_strength[0] = 0</td><td></td></tr> <tr><td> cdef_y_sec_strength[0] = 0</td><td></td></tr> <tr><td> cdef_uv_pri_strength[0] = 0</td><td></td></tr> <tr><td> cdef_uv_sec_strength[0] = 0</td><td></td></tr> <tr><td> CdefDamping = 3</td><td></td></tr> <tr><td> return</td><td></td></tr> <tr><td> }</td><td></td></tr> <tr><td> cdef_damping_minus_3</td><td>f(2)</td></tr> <tr><td> CdefDamping = cdef_damping_minus_3 + 3</td><td></td></tr> <tr><td> cdef_bits</td><td>f(2)</td></tr> <tr><td> for (i = 0; i < (1 << cdef_bits); i++) {</td><td></td></tr> <tr><td> cdef_y_pri_strength[i]</td><td>f(4)</td></tr> <tr><td> cdef_y_sec_strength[i]</td><td>f(2)</td></tr> <tr><td> if (cdef_y_sec_strength[i] == 3)</td><td></td></tr> <tr><td> cdef_y_sec_strength[i] += 1</td><td></td></tr> <tr><td> if (NumPlanes > 1) {</td><td></td></tr> <tr><td> cdef_uv_pri_strength[i]</td><td>f(4)</td></tr> <tr><td> cdef_uv_sec_strength[i]</td><td>f(2)</td></tr> <tr><td> if (cdef_uv_sec_strength[i] == 3)</td><td></td></tr> <tr><td> cdef_uv_sec_strength[i] += 1</td><td></td></tr> <tr><td> }</td><td></td></tr> <tr><td> }</td><td></td></tr> <tr><td>}</td><td></td></tr> </table> <u>The semantics of the CDEF params syntax are described in § 6.10.14:</u> S1: § 6.10.14. CDEF params semantics cdef_damping_minus_3 controls the amount of damping in the deringing filter.		Type	cdef_params() {		if (CodedLossless allow_intrabc		!enable_cdef) {		cdef_bits = 0		cdef_y_pri_strength[0] = 0		cdef_y_sec_strength[0] = 0		cdef_uv_pri_strength[0] = 0		cdef_uv_sec_strength[0] = 0		CdefDamping = 3		return		}		cdef_damping_minus_3	f(2)	CdefDamping = cdef_damping_minus_3 + 3		cdef_bits	f(2)	for (i = 0; i < (1 << cdef_bits); i++) {		cdef_y_pri_strength[i]	f(4)	cdef_y_sec_strength[i]	f(2)	if (cdef_y_sec_strength[i] == 3)		cdef_y_sec_strength[i] += 1		if (NumPlanes > 1) {		cdef_uv_pri_strength[i]	f(4)	cdef_uv_sec_strength[i]	f(2)	if (cdef_uv_sec_strength[i] == 3)		cdef_uv_sec_strength[i] += 1		}		}		}	
	Type																																																								
cdef_params() {																																																									
if (CodedLossless allow_intrabc																																																									
!enable_cdef) {																																																									
cdef_bits = 0																																																									
cdef_y_pri_strength[0] = 0																																																									
cdef_y_sec_strength[0] = 0																																																									
cdef_uv_pri_strength[0] = 0																																																									
cdef_uv_sec_strength[0] = 0																																																									
CdefDamping = 3																																																									
return																																																									
}																																																									
cdef_damping_minus_3	f(2)																																																								
CdefDamping = cdef_damping_minus_3 + 3																																																									
cdef_bits	f(2)																																																								
for (i = 0; i < (1 << cdef_bits); i++) {																																																									
cdef_y_pri_strength[i]	f(4)																																																								
cdef_y_sec_strength[i]	f(2)																																																								
if (cdef_y_sec_strength[i] == 3)																																																									
cdef_y_sec_strength[i] += 1																																																									
if (NumPlanes > 1) {																																																									
cdef_uv_pri_strength[i]	f(4)																																																								
cdef_uv_sec_strength[i]	f(2)																																																								
if (cdef_uv_sec_strength[i] == 3)																																																									
cdef_uv_sec_strength[i] += 1																																																									
}																																																									
}																																																									
}																																																									

CLAIM	PROTOCOL
	Cdef_bits specifies the number of bits needed to specify which CDEF filter to apply. Cdef_y_pri_strength and cdef_uv_pri_strength specify the strength of the primary filter. Cdef_y_sec_strength and cdef_uv_sec_strength specify the strength of the secondary filter. <u>As described above in § 7.15.1, the ordered steps of the CDEF filter process utilize the outputs yDir and var of the CDEF direction process, described below in § 7.15.2:</u> S1: § 7.15.2 CDEF direction process The inputs to this process are variables r and c specifying the location of an 8x8 block in units of 4x4 blocks in the luma plane. The outputs of this process are: • a variable yDir containing the direction of this block, • a variable var containing the variance for this block. This block uses luma samples to measure the direction and variance of a block. The process is specified as:

CLAIM	PROTOCOL
	<pre> for (i = 0; i < 8; i++) { cost[i] = 0 for (j = 0; j < 15; j++) partial[i][j] = 0 } bestCost = 0 yDir = 0 x0 = c << MI_SIZE_LOG2 y0 = r << MI_SIZE_LOG2 for (i = 0; i < 8; i++) { for (j = 0; j < 8; j++) { x = (CurrFrame[0][y0 + i][x0 + j] >> (BitDepth - 8)) - 128 partial[0][i + j] += x partial[1][i + j / 2] += x partial[2][i] += x partial[3][3 + i - j / 2] += x partial[4][7 + i - j] += x partial[5][3 - i / 2 + j] += x partial[6][j] += x partial[7][i / 2 + j] += x } } for (i = 0; i < 8; i++) { cost[2] += partial[2][i] * partial[2][i] cost[6] += partial[6][i] * partial[6][i] } cost[2] *= Div_Table[8] cost[6] *= Div_Table[8] for (i = 0; i < 7; i++) { cost[0] += (partial[0][i] * partial[0][i] + partial[0][14 - i] * partial[0][14 - i]) * Div_Table[i + 1] </pre>

CLAIM	PROTOCOL
	<pre> cost[4] += (partial[4][i] * partial[4][i] + partial[4][14 - i] * partial[4][14 - i]) * Div_Table[i + 1] } cost[0] += partial[0][7] * partial[0][7] * Div_Table[8] cost[4] += partial[4][7] * partial[4][7] * Div_Table[8] for (i = 1; i < 8; i += 2) { for (j = 0; j < 4 + 1; j++) { cost[i] += partial[i][3 + j] * partial[i][3 + j] } cost[i] *= Div_Table[8] for (j = 0; j < 4 - 1; j++) { cost[i] += (partial[i][j] * partial[i][j] + partial[i][10 - j] * partial[i][10 - j]) * Div_Table[2 * j + 2] } } for (i = 0; i < 8; i++) { if (cost[i] > bestCost) { bestCost = cost[i] yDir = i } } var = (bestCost - cost[(yDir + 4) & 7]) >> 10 </pre> where the Div_Table is a constant lookup table specified as:

CLAIM	PROTOCOL
	<pre>Div_Table[9] = { 0, 840, 420, 280, 210, 168, 140, 120, 105 }</pre> The adaptive sparse CDEF filter process invoked by the ordered steps in the CDEF block process above is applied as a second pass to reduce quantization noise by removing coding artifacts, and is described in § 7.15.3, below. <i>See, e.g.</i>, Midstskogen, et. al, “The AV1 Constrained Directional Enhancement Filter (CDEF),” April 2018, Conference: ICASSP 2018 - 2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), at 3 (“CDEF uses a non-linear low-pass filter designed to remove coding artifacts without blurring sharp edges. It achieves this by selecting filter tap locations based on the identified direction, but also by preventing excessive blurring when the filter is applied across an edge.”); <i>see also id.</i>, 1 (“The main goal of the in-loop constrained directional enhancement filter (CDEF) is to filter out coding artifacts while retaining the details of the image The amount of ringing artifacts in a coded image tends to be roughly proportional to the quantization step size. The amount of detail is a property of the input image, but the smallest detail actually retained in the quantized image tends to also be proportional to the quantization step size. For a given quantization step size, the amplitude of the ringing is generally less than the amplitude of the details. CDEF works by identifying the direction [4] of each block (Sec. 2) and then adaptively filtering along the identified direction (Sec. 3)”). S1: §7.15.3. CDEF filter process The inputs to this process are: • a variable plane specifying which plane is being predicted, • variables r and c specifying the location of an 8x8 block in units of 4x4 blocks in the luma plane, • a variable priStr specifying the primary filter strength, • a variable secStr specifying the secondary filter strength, • a variable damping specifying a shift used for damping, • a variable dir specifying the detected direction of the block. The process modifies samples in CdefFrame based on filtering samples from CurrFrame.

Exhibit 5 to InterDigital's Preliminary Infringement Contentions

CLAIM	PROTOCOL
	MiColStart, MiRowStart, MiColEnd, MiRowEnd are set equal to the values they had when the syntax element MiSizes[r][c] was written. (...) The filtering is applied as follows: (...)

```

subX = (plane > 0) ? subsampling_x : 0
subY = (plane > 0) ? subsampling_y : 0
x0 = (c * MI_SIZE) >> subX
y0 = (r * MI_SIZE) >> subY
w = 8 >> subX
h = 8 >> subY
for ( i = 0; i < h; i++ ) {
    for ( j = 0; j < w; j++ ) {
        sum = 0
        x = CurrFrame[plane][y0 + i][x0 + j]
        max = x
        min = x
        for ( k = 0; k < 2; k++ ) {
            for ( sign = -1; sign <= 1; sign += 2 ) {
                p = cdef_get_at(plane, x0, y0, i, j, dir, k, sign, subX, subY)
                if ( CdefAvailable ) {
                    sum += Cdef_Pri_Taps[(priStr >> coeffShift) & 1][k] * constrain(p - x, priStr,
damping)

                    max = Max(p, max)
                    min = Min(p, min)
                }
                for ( dirOff = -2; dirOff <= 2; dirOff += 4 ) {
                    s = cdef_get_at(plane, x0, y0, i, j, (dir + dirOff) & 7, k, sign, subX, subY)
                    if ( CdefAvailable ) {
                        sum += Cdef_Sec_Taps[(priStr >> coeffShift) & 1][k] * constrain(s - x,
secStr, damping)

                        max = Max(s, max)
                        min = Min(s, min)
                    }
                }
            }
        }
        CdefFrame[plane][y0 + i][x0 + j] = Clip3(min, max, x + ((8 + sum - (sum < 0)) >> 4) )
    }
}

```

CLAIM	PROTOCOL
	where Cdef_Pri_Taps and Cdef_Sec_Taps are constant lookup tables specified as: <pre> Cdef_Pri_Taps[2][2] = { { 4, 2 }, { 3, 3 } } Cdef_Sec_Taps[2][2] = { { 2, 1 }, { 2, 1 } } </pre> constrain is specified as: <pre> constrain(diff, threshold, damping) { if (!threshold) return 0 dampingAdj = Max(0, damping - FloorLog2(threshold)) sign = (diff < 0) ? -1 : 1 return sign * Clip3(0, Abs(diff), threshold - (Abs(diff) >> dampingAdj)) } </pre> (...) where Cdef_Directions is a constant lookup table defined as:

CLAIM	PROTOCOL
	<pre> Cdef_Directions[8][2][2] = { { { -1, 1 }, { -2, 2 } }, { { 0, 1 }, { -1, 2 } }, { { 0, 1 }, { 0, 2 } }, { { 0, 1 }, { 1, 2 } }, { { 1, 1 }, { 2, 2 } }, { { 1, 0 }, { 2, 1 } }, { { 1, 0 }, { 2, 0 } }, { { 1, 0 }, { 2, -1 } } } </pre>
[1.5] wherein the adaptive sparse de-noising filter is selectively enabled or disabled at a given level, the given level being at least a macroblock level, a slice level, a picture level or a sequence level.	The Accused Products either literally and/or under the doctrine of equivalents practice the element “wherein the adaptive sparse de-noising filter is selectively enabled or disabled at a given level, the given level being at least a macroblock level, a slice level, a picture level or a sequence level” via the AV1 Protocol, as described by at least the following technical specification(s). For example, the Accused Products identified in Appendix 1 practice the AV1 Specification, and, as shown below, the AV1 Specification specifies that the CDEF adaptive sparse de-noising filter is selectively enabled or disabled at a given level. The AV 1 Specification further describes that the CDEF filter is enabled or disabled at least at a macroblock level, a slice level, a picture level or a sequence level. Accordingly, the Accused Products, through operability with the AV1 Specification, practice Claim Element 1.5. <u>The AV1 bitstream uses the general OBU syntax as described in § 5.3, which includes the sequence_header_obu function.</u> S1: § 5.3. OBU syntax S1: § 5.3.1. General OBU syntax

CLAIM	PROTOCOL																																												
	<table> <tr> <td>open_bitstream_unit(sz) {</td><td>Type</td></tr> <tr> <td> obu_header()</td><td></td></tr> <tr> <td> if (obu_has_size_field) {</td><td></td></tr> <tr> <td> obu_size</td><td>leb128()</td></tr> <tr> <td> } else {</td><td></td></tr> <tr> <td> obu_size = sz - 1 - obu_extension_flag</td><td></td></tr> <tr> <td> }</td><td></td></tr> <tr> <td> startPosition = get_position()</td><td></td></tr> <tr> <td> if (obu_type != OBU_SEQUENCE_HEADER &&</td><td></td></tr> <tr> <td> obu_type != OBU_TEMPORAL_DELIMITER &&</td><td></td></tr> <tr> <td> OperatingPointIdc != 0 &&</td><td></td></tr> <tr> <td> obu_extension_flag == 1)</td><td></td></tr> <tr> <td> {</td><td></td></tr> <tr> <td> inTemporalLayer = (OperatingPointIdc >> temporal_id) & 1</td><td></td></tr> <tr> <td> inSpatialLayer = (OperatingPointIdc >> (spatial_id + 8)) & 1</td><td></td></tr> <tr> <td> if (!inTemporalLayer ! inSpatialLayer) {</td><td></td></tr> <tr> <td> drop_obu()</td><td></td></tr> <tr> <td> return</td><td></td></tr> <tr> <td> }</td><td></td></tr> <tr> <td> }</td><td></td></tr> <tr> <td> if (obu_type == OBU_SEQUENCE_HEADER)</td><td></td></tr> <tr> <td> sequence_header_obu()</td><td></td></tr> </table> The syntax structure for the AV1 sequence_header_obu function includes “enable_cdef,” as is described in § 5.5, below: S1: § 5.5. Sequence header OBU syntax S1: § 5.5.1. General	open_bitstream_unit(sz) {	Type	obu_header()		if (obu_has_size_field) {		obu_size	leb128()	} else {		obu_size = sz - 1 - obu_extension_flag		}		startPosition = get_position()		if (obu_type != OBU_SEQUENCE_HEADER &&		obu_type != OBU_TEMPORAL_DELIMITER &&		OperatingPointIdc != 0 &&		obu_extension_flag == 1)		{		inTemporalLayer = (OperatingPointIdc >> temporal_id) & 1		inSpatialLayer = (OperatingPointIdc >> (spatial_id + 8)) & 1		if (!inTemporalLayer ! inSpatialLayer) {		drop_obu()		return		}		}		if (obu_type == OBU_SEQUENCE_HEADER)		sequence_header_obu()	
open_bitstream_unit(sz) {	Type																																												
obu_header()																																													
if (obu_has_size_field) {																																													
obu_size	leb128()																																												
} else {																																													
obu_size = sz - 1 - obu_extension_flag																																													
}																																													
startPosition = get_position()																																													
if (obu_type != OBU_SEQUENCE_HEADER &&																																													
obu_type != OBU_TEMPORAL_DELIMITER &&																																													
OperatingPointIdc != 0 &&																																													
obu_extension_flag == 1)																																													
{																																													
inTemporalLayer = (OperatingPointIdc >> temporal_id) & 1																																													
inSpatialLayer = (OperatingPointIdc >> (spatial_id + 8)) & 1																																													
if (!inTemporalLayer ! inSpatialLayer) {																																													
drop_obu()																																													
return																																													
}																																													
}																																													
if (obu_type == OBU_SEQUENCE_HEADER)																																													
sequence_header_obu()																																													

CLAIM	PROTOCOL																										
	<table> <tr> <td>sequence_header_obu() {</td><td>Type</td></tr> <tr> <td> seq_profile</td><td>f(3)</td></tr> <tr> <td> still_picture</td><td>f(1)</td></tr> <tr> <td> reduced_still_picture_header</td><td>f(1)</td></tr> <tr> <td> if (reduced_still_picture_header) {</td><td></td></tr> <tr> <td>(...)</td><td></td></tr> <tr> <td> }</td><td></td></tr> <tr> <td> enable_superres</td><td>f(1)</td></tr> <tr> <td> enable_cdef</td><td>f(1)</td></tr> <tr> <td> enable_restoration</td><td>f(1)</td></tr> <tr> <td> color_config()</td><td></td></tr> <tr> <td> film_grain_params_present</td><td>f(1)</td></tr> <tr> <td>}</td><td></td></tr> </table> <u>The semantics of the sequence_header_obu syntax, including “enable_cdef”, are described in § 6.4, below:</u> S1: § 6.4. Sequence header OBU semantics S1: § 6.4.1. General sequence header OBU semantics (...) enable_cdef equal to 1 specifies that cdef filtering may be enabled. enable_cdef equal to 0 specifies that cdef filtering is disabled. Note: It is allowed to set enable_cdef equal to 1 even when cdef filtering is not used on any frame in the coded video sequence. <u>As part of the tile group OBU syntax, § 5.11, the decode block syntax is provided, which includes the mode_info function</u> S1: § 5.11.5. Decode block syntax	sequence_header_obu() {	Type	seq_profile	f(3)	still_picture	f(1)	reduced_still_picture_header	f(1)	if (reduced_still_picture_header) {		(...)		}		enable_superres	f(1)	enable_cdef	f(1)	enable_restoration	f(1)	color_config()		film_grain_params_present	f(1)	}	
sequence_header_obu() {	Type																										
seq_profile	f(3)																										
still_picture	f(1)																										
reduced_still_picture_header	f(1)																										
if (reduced_still_picture_header) {																											
(...)																											
}																											
enable_superres	f(1)																										
enable_cdef	f(1)																										
enable_restoration	f(1)																										
color_config()																											
film_grain_params_present	f(1)																										
}																											

	Type
decode_block(r, c, subSize) {	
MiRow = r	
MiCol = c	
MiSize = subSize	
bw4 = Num_4x4_Blocks_Wide[subSize]	
bh4 = Num_4x4_Blocks_High[subSize]	
if (bh4 == 1 && subsampling_y && (MiRow & 1) == 0)	
HasChroma = 0	
else if (bw4 == 1 && subsampling_x && (MiCol & 1) == 0)	
HasChroma = 0	
else	
HasChroma = NumPlanes > 1	
AvailU = is_inside(r - 1, c)	
AvailL = is_inside(r, c - 1)	
AvailUChroma = AvailU	
AvailLChroma = AvailL	
if (HasChroma) {	
if (subsampling_y && bh4 == 1)	
AvailUChroma = is_inside(r - 2, c)	
if (subsampling_x && bw4 == 1)	
AvailLChroma = is_inside(r, c - 2)	
} else {	
AvailUChroma = 0	
AvailLChroma = 0	
}	
mode_info()	
palette_tokens()	
read_block_tx_size()	
if (skip)	
reset_block_context(bw4, bh4)	
isCompound = RefFrame[1] > INTRA_FRAME	

```

for ( y = 0; y < bh4; y++ ) {
    for ( x = 0; x < bw4; x++ ) {
        YModes [ r + y ][ c + x ] = YMode
        if ( RefFrame[ 0 ] == INTRA_FRAME && HasChroma )
            UVModes [ r + y ][ c + x ] = UVMode
        for ( refList = 0; refList < 2; refList++ )
            RefFrames[ r + y ][ c + x ][ refList ] = RefFrame[ refList ]
        if ( is_inter ) {
            if ( !use_intrabc ) {
                CompGroupIdxs[ r + y ][ c + x ] = comp_group_idx
                CompoundIdxs[ r + y ][ c + x ] = compound_idx
            }
            for ( dir = 0; dir < 2; dir++ ) {
                InterpFilters[ r + y ][ c + x ][ dir ] = interp_filter[ dir ]
            }
            for ( refList = 0; refList < 1 + isCompound; refList++ ) {
                Mvs[ r + y ][ c + x ][ refList ] = Mv[ refList ]
            }
        }
    }
}

compute_prediction( )
residual( )
for ( y = 0; y < bh4; y++ ) {
    for ( x = 0; x < bw4; x++ ) {
        IsInters[ r + y ][ c + x ] = is_inter
        SkipModes[ r + y ][ c + x ] = skip_mode
        Skips[ r + y ][ c + x ] = skip
        TxSizes[ r + y ][ c + x ] = TxSize
        MiSizes[ r + y ][ c + x ] = MiSize
        SegmentIds[ r + y ][ c + x ] = segment_id
        PaletteSizes[ 0 ][ r + y ][ c + x ] = PaletteSizeY
    }
}

```

CLAIM	PROTOCOL																																
	<table> <tr><td>PaletteSizes[1][r + y][c + x] = PaletteSizeUV</td><td></td></tr> <tr><td>for (i = 0; i < PaletteSizeY; i++)</td><td></td></tr> <tr><td> PaletteColors[0][r + y][c + x][i] = palette_colors_y[i]</td><td></td></tr> <tr><td>for (i = 0; i < PaletteSizeUV; i++)</td><td></td></tr> <tr><td> PaletteColors[1][r + y][c + x][i] = palette_colors_u[i]</td><td></td></tr> <tr><td>for (i = 0; i < FRAME_LF_COUNT; i++)</td><td></td></tr> <tr><td> DeltaLFs[r + y][c + x][i] = DeltaLF[i]</td><td></td></tr> <tr><td> }</td><td></td></tr> <tr><td> }</td><td></td></tr> <tr><td>}</td><td></td></tr> </table> The mode_info function called in the above decode block is described in § 5.11.6, below: S1: §5.11.6. Mode info syntax <table> <tr><td>mode_info() {</td><td>Type</td></tr> <tr><td> if (FrameIsIntra)</td><td></td></tr> <tr><td> intra_frame_mode_info()</td><td></td></tr> <tr><td> else</td><td></td></tr> <tr><td> inter_frame_mode_info()</td><td></td></tr> <tr><td> }</td><td></td></tr> </table> When mode_info is not FrameIsIntra, the inter_frame_mode_info function is called, as described in § 5.11.P8, below: S1: § 5.11.P8. Inter frame mode info syntax	PaletteSizes[1][r + y][c + x] = PaletteSizeUV		for (i = 0; i < PaletteSizeY; i++)		PaletteColors[0][r + y][c + x][i] = palette_colors_y[i]		for (i = 0; i < PaletteSizeUV; i++)		PaletteColors[1][r + y][c + x][i] = palette_colors_u[i]		for (i = 0; i < FRAME_LF_COUNT; i++)		DeltaLFs[r + y][c + x][i] = DeltaLF[i]		}		}		}		mode_info() {	Type	if (FrameIsIntra)		intra_frame_mode_info()		else		inter_frame_mode_info()		}	
PaletteSizes[1][r + y][c + x] = PaletteSizeUV																																	
for (i = 0; i < PaletteSizeY; i++)																																	
PaletteColors[0][r + y][c + x][i] = palette_colors_y[i]																																	
for (i = 0; i < PaletteSizeUV; i++)																																	
PaletteColors[1][r + y][c + x][i] = palette_colors_u[i]																																	
for (i = 0; i < FRAME_LF_COUNT; i++)																																	
DeltaLFs[r + y][c + x][i] = DeltaLF[i]																																	
}																																	
}																																	
}																																	
mode_info() {	Type																																
if (FrameIsIntra)																																	
intra_frame_mode_info()																																	
else																																	
inter_frame_mode_info()																																	
}																																	

CLAIM	PROTOCOL																																																														
	<table> <tr> <th></th><th>Type</th></tr> <tr> <td>inter_frame_mode_info() {</td><td></td></tr> <tr> <td> use_intrabc = 0</td><td></td></tr> <tr> <td> LeftRefFrame[0] = AvailL ? RefFrames[MiRow][MiCol-1][0] : INTRA_FRAME</td><td></td></tr> <tr> <td> AboveRefFrame[0] = AvailU ? RefFrames[MiRow-1][MiCol][0] : INTRA_FRAME</td><td></td></tr> <tr> <td> LeftRefFrame[1] = AvailL ? RefFrames[MiRow][MiCol-1][1] : NONE</td><td></td></tr> <tr> <td> AboveRefFrame[1] = AvailU ? RefFrames[MiRow-1][MiCol][1] : NONE</td><td></td></tr> <tr> <td> LeftIntra = LeftRefFrame[0] <= INTRA_FRAME</td><td></td></tr> <tr> <td> AboveIntra = AboveRefFrame[0] <= INTRA_FRAME</td><td></td></tr> <tr> <td> LeftSingle = LeftRefFrame[1] <= INTRA_FRAME</td><td></td></tr> <tr> <td> AboveSingle = AboveRefFrame[1] <= INTRA_FRAME</td><td></td></tr> <tr> <td> skip = 0</td><td></td></tr> <tr> <td> inter_segment_id(1)</td><td></td></tr> <tr> <td> read_skip_mode()</td><td></td></tr> <tr> <td> if (skip_mode)</td><td></td></tr> <tr> <td> skip = 1</td><td></td></tr> <tr> <td> else</td><td></td></tr> <tr> <td> read_skip()</td><td></td></tr> <tr> <td> if (!SegIdPreSkip)</td><td></td></tr> <tr> <td> inter_segment_id(0)</td><td></td></tr> <tr> <td> Lossless = LosslessArray[segment_id]</td><td></td></tr> <tr> <td> read_cdef()</td><td></td></tr> <tr> <td> read_delta_qindex()</td><td></td></tr> <tr> <td> read_delta_lf()</td><td></td></tr> <tr> <td> ReadDeltas = 0</td><td></td></tr> <tr> <td> read_is_inter()</td><td></td></tr> <tr> <td> if (is_inter)</td><td></td></tr> <tr> <td> inter_block_mode_info()</td><td></td></tr> <tr> <td> else</td><td></td></tr> <tr> <td> intra_block_mode_info()</td><td></td></tr> <tr> <td> }</td><td></td></tr> </table>		Type	inter_frame_mode_info() {		use_intrabc = 0		LeftRefFrame[0] = AvailL ? RefFrames[MiRow][MiCol-1][0] : INTRA_FRAME		AboveRefFrame[0] = AvailU ? RefFrames[MiRow-1][MiCol][0] : INTRA_FRAME		LeftRefFrame[1] = AvailL ? RefFrames[MiRow][MiCol-1][1] : NONE		AboveRefFrame[1] = AvailU ? RefFrames[MiRow-1][MiCol][1] : NONE		LeftIntra = LeftRefFrame[0] <= INTRA_FRAME		AboveIntra = AboveRefFrame[0] <= INTRA_FRAME		LeftSingle = LeftRefFrame[1] <= INTRA_FRAME		AboveSingle = AboveRefFrame[1] <= INTRA_FRAME		skip = 0		inter_segment_id(1)		read_skip_mode()		if (skip_mode)		skip = 1		else		read_skip()		if (!SegIdPreSkip)		inter_segment_id(0)		Lossless = LosslessArray[segment_id]		read_cdef()		read_delta_qindex()		read_delta_lf()		ReadDeltas = 0		read_is_inter()		if (is_inter)		inter_block_mode_info()		else		intra_block_mode_info()		}	
	Type																																																														
inter_frame_mode_info() {																																																															
use_intrabc = 0																																																															
LeftRefFrame[0] = AvailL ? RefFrames[MiRow][MiCol-1][0] : INTRA_FRAME																																																															
AboveRefFrame[0] = AvailU ? RefFrames[MiRow-1][MiCol][0] : INTRA_FRAME																																																															
LeftRefFrame[1] = AvailL ? RefFrames[MiRow][MiCol-1][1] : NONE																																																															
AboveRefFrame[1] = AvailU ? RefFrames[MiRow-1][MiCol][1] : NONE																																																															
LeftIntra = LeftRefFrame[0] <= INTRA_FRAME																																																															
AboveIntra = AboveRefFrame[0] <= INTRA_FRAME																																																															
LeftSingle = LeftRefFrame[1] <= INTRA_FRAME																																																															
AboveSingle = AboveRefFrame[1] <= INTRA_FRAME																																																															
skip = 0																																																															
inter_segment_id(1)																																																															
read_skip_mode()																																																															
if (skip_mode)																																																															
skip = 1																																																															
else																																																															
read_skip()																																																															
if (!SegIdPreSkip)																																																															
inter_segment_id(0)																																																															
Lossless = LosslessArray[segment_id]																																																															
read_cdef()																																																															
read_delta_qindex()																																																															
read_delta_lf()																																																															
ReadDeltas = 0																																																															
read_is_inter()																																																															
if (is_inter)																																																															
inter_block_mode_info()																																																															
else																																																															
intra_block_mode_info()																																																															
}																																																															

CLAIM	PROTOCOL																																				
	<u>The read_cdef function called in the above inter frame mode selectively enables or disables the CDEF filter using enable_cdef and cdef_idx, as described in § 5.11.56, below:</u> S1: § 5.11.56. Read CDEF syntax <table border="1"> <thead> <tr> <th>read_cdef() {</th><th>Type</th></tr> </thead> <tbody> <tr> <td>if (skip CodedLossless !enable_cdef allow_intrabc) {</td><td></td></tr> <tr> <td>return</td><td></td></tr> <tr> <td>}</td><td></td></tr> <tr> <td>cdefSize4 = Num_4x4_Blocks_Wide[BLOCK_64X64]</td><td></td></tr> <tr> <td>cdefMask4 = ~(cdefSize4 - 1)</td><td></td></tr> <tr> <td>r = MiRow & cdefMask4</td><td></td></tr> <tr> <td>c = MiCol & cdefMask4</td><td></td></tr> <tr> <td>if (cdef_idx[r][c] == -1) {</td><td></td></tr> <tr> <td>cdef_idx[r][c]</td><td>L(cdef_bits)</td></tr> <tr> <td>w4 = Num_4x4_Blocks_Wide[MiSize]</td><td></td></tr> <tr> <td>h4 = Num_4x4_Blocks_High[MiSize]</td><td></td></tr> <tr> <td>for (i = r; i < r + h4 ; i += cdefSize4) {</td><td></td></tr> <tr> <td>for (j = c; j < c + w4 ; j += cdefSize4) {</td><td></td></tr> <tr> <td>cdef_idx[i][j] = cdef_idx[r][c]</td><td></td></tr> <tr> <td>}</td><td></td></tr> <tr> <td>}</td><td></td></tr> <tr> <td>}</td><td></td></tr> </tbody> </table> <u>The semantics for the Read CDEF function is described in § 6.10.40, below:</u> S1: § 6.10.40. Read CDEF semantics cdef_idx specifies which CDEF filtering parameters should be used for a particular 64 by 64 block. A value of -1 means that CDEF is disabled for that block.	read_cdef() {	Type	if (skip CodedLossless !enable_cdef allow_intrabc) {		return		}		cdefSize4 = Num_4x4_Blocks_Wide[BLOCK_64X64]		cdefMask4 = ~(cdefSize4 - 1)		r = MiRow & cdefMask4		c = MiCol & cdefMask4		if (cdef_idx[r][c] == -1) {		cdef_idx[r][c]	L(cdef_bits)	w4 = Num_4x4_Blocks_Wide[MiSize]		h4 = Num_4x4_Blocks_High[MiSize]		for (i = r; i < r + h4 ; i += cdefSize4) {		for (j = c; j < c + w4 ; j += cdefSize4) {		cdef_idx[i][j] = cdef_idx[r][c]		}		}		}	
read_cdef() {	Type																																				
if (skip CodedLossless !enable_cdef allow_intrabc) {																																					
return																																					
}																																					
cdefSize4 = Num_4x4_Blocks_Wide[BLOCK_64X64]																																					
cdefMask4 = ~(cdefSize4 - 1)																																					
r = MiRow & cdefMask4																																					
c = MiCol & cdefMask4																																					
if (cdef_idx[r][c] == -1) {																																					
cdef_idx[r][c]	L(cdef_bits)																																				
w4 = Num_4x4_Blocks_Wide[MiSize]																																					
h4 = Num_4x4_Blocks_High[MiSize]																																					
for (i = r; i < r + h4 ; i += cdefSize4) {																																					
for (j = c; j < c + w4 ; j += cdefSize4) {																																					
cdef_idx[i][j] = cdef_idx[r][c]																																					
}																																					
}																																					
}																																					

CLAIM	PROTOCOL
	<u>The CDEF process, including input cdef_idx, is then described in § 7.15, below:</u> S1: § 7.15. CDEF process Input to this process is the array CurrFrame of reconstructed samples. Output from this process is the array CdefFrame containing deringed samples. The purpose of CDEF is to perform deringing based on the detected direction of blocks. CDEF parameters are stored for each 64 by 64 block of luma samples. The CDEF filter is applied on each 8 by 8 block as follows: <pre> step4 = Num_4x4_Blocks_Wide[BLOCK_8X8] cdefSize4 = Num_4x4_Blocks_Wide[BLOCK_64X64] cdefMask4 = ~(cdefSize4 - 1) for (r = 0; r < MiRows; r += step4) { for (c = 0; c < MiCols; c += step4) { baseR = r & cdefMask4 baseC = c & cdefMask4 idx = cdef_idx[baseR][baseC] cdef_block(r, c, idx) } } </pre> When the cdef_block function is called, the CDEF block process specified in section 7.15.1 is invoked with r, c, and idx as inputs. S1: § 7.15.1. CDEF block process The inputs to this process are: • variables r and c specifying the location of an 8x8 block in units of 4x4 blocks in the luma plane, • a variable idx specifying which set of CDEF parameters to use, or -1 to signal that no filtering should be applied.

CLAIM	PROTOCOL
2. The method of claim 1, wherein at least one of the at least two filters is used to enforce at least one compression constraint.	Given the Accused Products decode according to AV1, upon information and belief and subject to further discovery, the Accused Products likely either literally and/or under the doctrine of equivalents practice the claim “wherein at least one of the at least two filters is used to enforce at least one compression constraint.” <i>See</i> Claim 1, which describes the underlying independent claim.
3. The method of claim 1, wherein at least one of the at least two filters is adaptively enabled or disabled depending upon at least data characteristics corresponding to the image region or data statistics corresponding to the image region.	The Accused Products either literally and/or under the doctrine of equivalents practice the claim “at least one of the at least two filters is adaptively enabled or disabled depending upon at least data characteristics corresponding to the image region” via the AV1 Protocol, as described by at least the following technical specification(s). For example, the Accused Products identified in Appendix 1 practice the AV1 Specification, and, as shown below, the AV1 Specification specifies that at least one of the filters is adaptively enabled or disabled depending on data corresponding to the image region. Accordingly, the Accused Products, through operability with the AV1 Specification, practice Claim 3. <i>See also</i> Claim 1, which describes the underlying independent claim. <u>As described above with respect to Claim Element 1.2, the AV1 general tile group OBU syntax table includes a call to perform in-loop filtering, and as described above in Claim Element 1.3, the filtering is performed by invoking the ordered steps of (1) the loop filter process as in § 7.14 and (2) the CDEF process as in § 7.15. AV1’s loop filter is adaptively enabled or disabled depending on data characteristics or data statistics corresponding to the image region.</u> S1: § 7.14. Loop filter process S1: § 7.14.1 General Input to this process is the array CurrFrame of reconstructed samples. Output from this process is a modified array CurrFrame containing deblocked samples. The purpose of the loop filter is to eliminate (or at least reduce) visually objectionable artifacts associated with the semi- independence of the coding of super blocks and their constituent sub-blocks. The loop filter is applied on all vertical boundaries followed by all horizontal boundaries as follows:

CLAIM	PROTOCOL
	<pre data-bbox="457 272 1900 755"> for (plane = 0; plane < NumPlanes; plane++) { if (plane == 0 loop_filter_level[1 + plane]) { for (pass = 0; pass < 2; pass++) { rowStep = (plane == 0) ? 1 : (1 << subsampling_y) colStep = (plane == 0) ? 1 : (1 << subsampling_x) for (row = 0; row < MiRows; row += rowStep) for (col = 0; col < MiCols; col += colStep) loop_filter_edge(plane, pass, row, col) } } } </pre> <u>The edge loop filter process called by the loop filter is described in § 7.14.2, below:</u> S1: § 7.14.2. Edge loop filter process The inputs to this process are: • a variable plane specifying whether the process is filtering Y, U, or V samples, • a variable pass specifying the direction of the edges. pass equal to 0 means the process is filtering vertical block boundaries, and pass equal to 1 means the process is filtering horizontal block boundaries, • variables row and col specifying the location of the edge in units of 4x4 blocks in the luma plane. The outputs of this process are modified values in the array CurrFrame. (…) The variable applyFilter (equal to 1 if the samples are filtered) is derived as follows: • If isTxEdge is equal to 0, applyFilter is set equal to 0.

CLAIM	PROTOCOL
	- Otherwise, if isBlockEdge is equal to 1 or skip is equal to 0 or isIntra is equal to 1, applyFilter is set equal to 1. - Otherwise applyFilter is set equal to 0. The filter size process specified in section 7.14.3 is invoked with the inputs txSz, prevTxSz, pass, and plane, and the output assigned to the variable filterSize (containing the maximum filter size that can be used). The adaptive filter strength process specified in section 7.14.4 is invoked with the inputs row, col, plane, and pass, and the output assigned to the variables lvl, limit, blimit, and thresh. If lvl is equal to 0, the adaptive filter strength process specified in section 7.14.4 is invoked with the inputs prevRow, prevCol, plane, and pass, and the output assigned to the variables lvl, limit, blimit, and thresh. For the variable i taking values from 0 to MI_SIZE - 1, the following applies: - If applyFilter is equal to 1 and lvl is greater than zero, the sample filtering process specified in section 7.14.6 is invoked with the input variable x set equal to $xP + dy * i$, the input variable y set equal to $yP + dx * i$, and the variables plane, limit, blimit, thresh, dx, dy, and filterSize supplied as inputs. <u>The sample filter process called in the above edge loop filter process is described in § 7.14.6, below:</u> S1: § 7.14.6. Sample filtering process S1: § 7.14.6.1. General First the filter mask process specified in section 7.14.6.2 is invoked with the inputs x, y, plane, limit, blimit, thresh, dx, dy, and filterSize, and the output is assigned to the variables hevMask, filterMask, flatMask, and flatMask2. Then the appropriate filter process is invoked with the inputs x, y, plane, dx, dy as follows: - If filterMask is equal to 0, no filter is invoked. - Otherwise, if filterSize is equal to 4 or flatMask is equal to 0, the narrow filter process specified in section 7.14.6.3 is invoked with the additional input variable hevMask.

CLAIM	PROTOCOL
	• Otherwise, if filterSize is equal to 8 or flatMask2 is equal to 0, the wide filter process specified in section 7.14.6.4 is invoked with the additional input variable log2Size set to 3. • Otherwise, the wide filter process specified in section 7.14.6.4 is invoked with the additional input variable log2Size set to 4. <u>The filter mask process invoked in the above sample filtering process is described in § 7.14.6.2, below:</u> S1: § 7.14.6.2. Filter mask process The inputs to this process are: • variables x and y specifying the location within CurrFrame[plane], • a variable plane specifying whether the block is the Y, U or V plane, • variables limit, blimit, thresh that specify the strength of the filtering operation, • variables dx and dy specifying the direction perpendicular to the edge being filtered, • a variable filterSize of specifying the maximum size of filter allowed. The outputs from this process are the variables: • hevMask, • filterMask, • flatMask, (only used if filterSize ≥ 8), • flatMask2 (only used if filterSize ≥ 16). The values output for these masks depend on the differences between samples on either side of the specified boundary. These samples are specified as follows:

CLAIM	PROTOCOL
	<pre> q0 = CurrFrame[plane][y][x] q1 = CurrFrame[plane][y + dy][x + dx] q2 = CurrFrame[plane][y + dy * 2][x + dx * 2] q3 = CurrFrame[plane][y + dy * 3][x + dx * 3] q4 = CurrFrame[plane][y + dy * 4][x + dx * 4] q5 = CurrFrame[plane][y + dy * 5][x + dx * 5] q6 = CurrFrame[plane][y + dy * 6][x + dx * 6] p0 = CurrFrame[plane][y - dy][x - dx] p1 = CurrFrame[plane][y - dy * 2][x - dx * 2] p2 = CurrFrame[plane][y - dy * 3][x - dx * 3] p3 = CurrFrame[plane][y - dy * 4][x - dx * 4] p4 = CurrFrame[plane][y - dy * 5][x - dx * 5] p5 = CurrFrame[plane][y - dy * 6][x - dx * 6] p6 = CurrFrame[plane][y - dy * 7][x - dx * 7] </pre> Note: Samples q4, q5, q6, p4, p5, and p6 are only used if filterSize is equal to 16. (...) The value of filterMask indicates whether adjacent samples close to the edge (within four samples either side of the specified boundary) vary by less than the limits given by limit and blimit. It is used to determine if any filtering should occur and is calculated as follows:

CLAIM	PROTOCOL
	<pre> limitBd = limit << (BitDepth - 8) blimitBd = blimit << (BitDepth - 8) mask = 0 mask = (Abs(p1 - p0) > limitBd) mask = (Abs(q1 - q0) > limitBd) mask = (Abs(p0 - q0) * 2 + Abs(p1 - q1) / 2 > blimitBd) if (filterLen >= 6) { mask = (Abs(p2 - p1) > limitBd) mask = (Abs(q2 - q1) > limitBd) } if (filterLen >= 8) { mask = (Abs(p3 - p2) > limitBd) mask = (Abs(q3 - q2) > limitBd) } filterMask = (mask == 0) </pre>
4. The method of claim 1, wherein a version of the image region that is displayed is different from at least one of an output signal corresponding to the image region after the at least two in-loop filters and an initial input signal corresponding to the image region before the at least two in-loop filters.	The Accused Products either literally and/or under the doctrine of equivalents practice the claim “a version of the image region that is displayed is different from at least one of an output signal corresponding to the image region after the at least two in-loop filters and an initial input signal corresponding to the image region before the at least two in-loop filters” via the AV1 Protocol, as described by at least the following technical specification(s). For example, the Accused Products identified in Appendix 1 practice the AV1 Specification, and, as shown below, the AV1 Specification has two operating modes: the first is general decoding where the input is a sequence of OBUs and the output is decoded frames, which is specified in the general decoding process of the protocol. The general decoding process § 7.2 then specifies that when <code>film_grain_params_present</code> is equal to 0, the decoders produce output frames according to the general decoding process detailed below, while the decoder output when <code>film_grains_params_present</code> is equal to 1 is modified using a film grain synthesis process as set forth below. Accordingly, the Accused Products, through operability with the AV1 Specification, practice Claim 4. <i>See also</i> Claim 1, which describes the underlying independent claim. <u>As described above with respect to Claim Element 1.1, the AV1 bitstream two operating modes: the first is general decoding where the input is a sequence of OBUs and the output is decoded frames, which is specified in the general decoding process of the protocol, as described in § 7.2, below:</u>

CLAIM	PROTOCOL
	S1: § 7.2. General decoding process When <code>film_grain_params_present</code> is equal to 0, decoders shall produce output frames that are identical in all respects and have the same output order as those produced by the decoding process specified herein. When <code>film_grain_params_present</code> is equal to 1, a decoder shall implement a film grain synthesis process that modifies the output arrays <code>OutY</code>, <code>OutU</code>, <code>OutV</code>. The reference film grain synthesis process is described in section 7.18.3. <u>As described above with respect to Claim Element 1.2, the <code>decode_frame_wrapup</code> process performs in-loop filtering using at least two filters in succession, relevant portions of the decode frame wrapup process are described in § 7.4, below:</u> S1: § 7.4. Decode frame wrapup process This process is triggered by a call to <code>decode_frame_wrapup</code> from within the syntax tables. At this stage, all the tile level decode has been done, and this process performs any frame level decode that is required. If <code>show_existing_frame</code> is equal to 0, the process first performs any post processing filtering by the following ordered steps: 1. If <code>loop_filter_level[0]</code> is not equal to 0 or <code>loop_filter_level[1]</code> is not equal to 0, the loop filter process specified in section 7.14 is invoked (this process modifies the contents of <code>CurrFrame</code>). 2. The CDEF process specified in section 7.15 is invoked (this process takes <code>CurrFrame</code> and produces <code>CdefFrame</code>). (...) The following ordered steps now apply: 1. The reference frame update process as specified in section 7.20 is invoked (this process saves the current frame state into the reference frames). 2. If <code>show_frame</code> is equal to 1 or <code>show_existing_frame</code> is equal to 1, the output process as specified in section 7.18 is invoked (this will output the current frame or a saved frame).

CLAIM	PROTOCOL
	<u>As described above with respect to Claim Element 1.2, the AV1 loop filter process is applied, and relevant portions of the loop filter process are described in § 7.14, below:</u> S1: § 7.14. Loop filter process S1: § 7.14.1. General Input to this process is the array CurrFrame of reconstructed samples. Output from this process is a modified array CurrFrame containing deblocked samples. <u>As described above with respect to Claim Element 1.3, the AV1 loop filter process is applied, and relevant portions of the CDEF process are described in § 7.15, below:</u> S1: § 7.15. CDEF process Input to this process is the array CurrFrame of reconstructed samples. Output from this process is the array CdefFrame containing deringed samples. <u>The output process called in the decode frame wrapup process when show_frame is equal to 1 or show_existing_frame is equal to 1 is described in § 7.18, below:</u> S1: § 7.18. Output process S1: § 7.18.1. General This process is invoked to prepare output frames. (...) The intermediate output preparation process specified in section 7.18.2 is invoked to prepare arrays OutY, OutU, and OutV, and the outputs are assigned to w, h, subX, and subY. If film_grain_params_present is equal to 1 and apply_grain is equal to 1, then the film grain synthesis process specified in section 7.18.3 is invoked with inputs of w, h, subX, and subY. (This process modifies the output arrays OutY, OutU,

CLAIM	PROTOCOL
	OutV). Finally, the frame to be output is defined to be the arrays OutY, OutU, OutV where the bit depth for each sample is BitDepth. This frame to be output is the overall output of the decoding process and further processing (such as color conversion) is outside the scope of this specification. For example, a real implementation might use these arrays to display the frame to the user, or a test system might save the arrays so the output can be verified.
[5.P] An apparatus comprising at least a memory and one or more processors, the one or more processors being configured to:	To the extent the preamble is limiting, the Accused Products either literally and/or under the doctrine of equivalents are an apparatus comprising at least a memory and one or more processors and comply with the AV1 Protocol, as described by at least the following technical specification(s): S1: Alliance for Open Media AV1 Version 1.0.0 with Errata 1 (2019-01-08 11:48 PT) The products identified in Appendix 1 herewith, are each an apparatus comprising at least a memory and one or more processors configured to decode video pursuant to the AV1 video coding protocol. <i>See also</i> IDG1373_0002182–2190 (Lenovo Website - Lenovo ThinkPad X1 Carbon Gen. 10); IDG1373_0003779–3795 (Intel website – 12th Gen Intel Mobile Product Brief, at 5, 8); IDG1373_0003718–3728 (Intel Website – Intel Processor Supported Codecs, at 6-7).
[5.1] decode an image region,	The Accused Products either literally and/or under the doctrine of equivalents practice the element “decode an image region,” via the AV1 Protocol, as described by at least the following technical specification(s). For example, the Accused Products identified in Appendix 1 practice the AV1 Specification, and, as shown below, the AV1 Specification specifies that the image region is decoded. Therefore, the Accused Products, through operability with the AV1 Specification, practice Claim Element 5.1. S1: § 1. Scope This document specifies the Alliance for Open Media AV1 bitstream formats and decoding process. <u>The Alliance for Open Media AV1 bitstream utilizes the below process for decoding an image region.</u>

CLAIM	PROTOCOL
	S1: § 7. Decoding process S1: § 7.1. Overview AV1 contains two operating modes: 1. General decoding (input is a sequence of OBUs, output is decoded frames) 2. Large scale tile decoding (input is a tile list OBU plus additional side information, output is a decoded frame) The general decoding process is specified in section 7.2. S1: § 7.2 General decoding process When <code>film_grain_params_present</code> is equal to 0, decoders shall produce output frames that are identical in all respects and have the same output order as those produced by the decoding process specified herein. (...) The input to this process is a sequence of open bitstream units (OBUs). The output from this process is a sequence of decoded frames.
[5.2] the one or more processors including at least two filters for successively performing in-loop filtering to reduce at least a first and a second type of coding artifact,	The Accused Products either literally and/or under the doctrine of equivalents practice the element “the one or more processors including at least two filters for successively performing in-loop filtering to reduce at least a first and a second type of coding artifact,” via the AV1 Protocol, as described by at least the following technical specification(s). For example, the Accused Products identified in Appendix 1 practice the AV1 Specification, and, as shown below, the AV1 Specification specifies that processor(s) include in-loop filtering that uses at least two filters in succession, a loop filter and a CDEF filter, to reduce coding artifacts when decoding the image region. Therefore, the Accused Products, through operability with the AV1 Specification, practice Claim Element 5.2. <u>In the AV1 specification, the frame is coded in the Frame OBU syntax table which includes a frame header called <code>frame_header_obu()</code> and the coded frame data coded in <code>tile_group_obu (sz)</code> syntax table.</u> S1: § 5.10. Frame OBU syntax

CLAIM	PROTOCOL																		
	<table border="1"> <thead> <tr> <th>frame_obu(sz) {</th><th>Type</th></tr> </thead> <tbody> <tr> <td> startBitPos = get_position()</td><td></td></tr> <tr> <td> frame_header_obu()</td><td></td></tr> <tr> <td> byte_alignment()</td><td></td></tr> <tr> <td> endBitPos = get_position()</td><td></td></tr> <tr> <td> headerBytes = (endBitPos - startBitPos) / 8</td><td></td></tr> <tr> <td> sz -= headerBytes</td><td></td></tr> <tr> <td> tile_group_obu(sz)</td><td></td></tr> <tr> <td>}</td><td></td></tr> </tbody> </table> S1: § 6.9. Frame OBU semantics A frame OBU consists of a frame header OBU and a tile group OBU packed into a single OBU. Note: The intention is to provide a more compact way of coding the common use case where the frame header is immediately followed by tile group data. S1: § 5.11. Tile group OBU syntax S1: § 5.11.P. General tile group OBU syntax	frame_obu(sz) {	Type	startBitPos = get_position()		frame_header_obu()		byte_alignment()		endBitPos = get_position()		headerBytes = (endBitPos - startBitPos) / 8		sz -= headerBytes		tile_group_obu(sz)		}	
frame_obu(sz) {	Type																		
startBitPos = get_position()																			
frame_header_obu()																			
byte_alignment()																			
endBitPos = get_position()																			
headerBytes = (endBitPos - startBitPos) / 8																			
sz -= headerBytes																			
tile_group_obu(sz)																			
}																			

Exhibit 5 to InterDigital's Preliminary Infringement Contentions

CLAIM	PROTOCOL	
	tile_group_obu(sz) {	Type
	NumTiles = TileCols * TileRows	
	startBitPos = get_position()	
	tile_start_and_end_present_flag = 0	
	if (NumTiles > 1)	
	tile_start_and_end_present_flag	f(1)
	if (NumTiles == 1 !tile_start_and_end_present_flag) {	
	tg_start = 0	
	tg_end = NumTiles - 1	
	} else {	
	tileBits = TileColsLog2 + TileRowsLog2	
	tg_start	f(tileBits)
	tg_end	f(tileBits)
	}	
	byte_alignment()	

endBitPos = get_position()	
headerBytes = (endBitPos - startBitPos) / 8	
sz -= headerBytes	
for (TileNum = tg_start; TileNum <= tg_end; TileNum++) {	
tileRow = TileNum / TileCols	
tileCol = TileNum % TileCols	
lastTile = TileNum == tg_end	
if (lastTile) {	
tileSize = sz	
} else {	
tile_size_minus_1	le(TileSizeBytes)
tileSize = tile_size_minus_1 + 1	
sz -= tileSize + TileSizeBytes	
}	
MiRowStart = MiRowStarts[tileRow]	
MiRowEnd = MiRowStarts[tileRow + 1]	
MiColStart = MiColStarts[tileCol]	
MiColEnd = MiColStarts[tileCol + 1]	
CurrentQIndex = base_q_idx	
init_symbol(tileSize)	
decode_tile()	
exit_symbol()	
}	
if (tg_end == NumTiles - 1) {	
if (!disable_frame_end_update_cdf) {	
frame_end_update_cdf()	
}	
decode_frame_wrapup()	
SeenFrameHeader = 0	
}	
}	

CLAIM	PROTOCOL
	<u>After the decoding of the tiles of the frame invoked from decode_tile (), the function decode_frame_wrapup () is invoked. The decode_frame_wrapup function of AV1 is described in § 6.8, below:</u> S1: § 6.10. Tile group OBU semantics S1: § 6.10.1. General tile group OBU semantics decode_frame_wrapup is a function call that indicates that the decode frame wrapup process specified in section 7.4 should be invoked. <u>The decode_frame_wrapup process to perform in-loop filtering to reduce at least a first and a second type of coding artifact using at least two filters in succession results from the decode_frame_wrapup function call, and is described in § 7.4, below:</u> S1: § 7.4. Decode frame wrapup process This process is triggered by a call to decode_frame_wrapup from within the syntax tables. At this stage, all the tile level decode has been done, and this process performs any frame level decode that is required. If show_existing_frame is equal to 0, the process first performs any post processing filtering by the following ordered steps: 1. If loop_filter_level[0] is not equal to 0 or loop_filter_level[1] is not equal to 0, the loop filter process specified in section 7.14 is invoked (this process modifies the contents of CurrFrame). 2. The CDEF process specified in section 7.15 is invoked (this process takes CurrFrame and produces CdefFrame). <u>AV1's loop filter process, the first of the at least two filters in succession, is specified in § 7.14, below:</u> S1: § 7.14. Loop filter process S1: § 7.14.1 General Input to this process is the array CurrFrame of reconstructed samples.

CLAIM	PROTOCOL
	Output from this process is a modified array CurrFrame containing deblocked samples. The purpose of the loop filter is to eliminate (or at least reduce) visually objectionable artifacts associated with the semi- independence of the coding of super blocks and their constituent sub-blocks. The loop filter is applied on all vertical boundaries followed by all horizontal boundaries as follows: <pre> for (plane = 0; plane < NumPlanes; plane++) { if (plane == 0 loop_filter_level[1 + plane]) { for (pass = 0; pass < 2; pass++) { rowStep = (plane == 0) ? 1 : (1 << subsampling_y) colStep = (plane == 0) ? 1 : (1 << subsampling_x) for (row = 0; row < MiRows; row += rowStep) for (col = 0; col < MiCols; col += colStep) loop_filter_edge(plane, pass, row, col) } } } </pre> <u>AV1's CDEF process, the second of the at least two filters in succession, is specified in § 7.15, below:</u> S1: § 7.15. CDEF process Input to this process is the array CurrFrame of reconstructed samples. Output from this process is the array CdefFrame containing deringed samples. The purpose of CDEF is to perform deringing based on the detected direction of blocks.
[5.3] the at least two filters including a	The Accused Products either literally and/or under the doctrine of equivalents practice the element “the at least two filters including a deblocking filter for performing a first pass to reduce blocking artifacts” via the AV1 Protocol, as described by at least the following technical

CLAIM	PROTOCOL
deblocking filter for performing a first pass to reduce blocking artifacts	specification(s). For example, the Accused Products identified in Appendix 1 practice the AV1 Specification, and, as shown below, the AV1 Specification specifies that a loop filter process is used to perform a first pass to reduce blocking artifacts and the output of this process is deblocked samples. Therefore, the loop filter described in the AV1 Specification is a deblocking filter for performing a first pass to reduce blocking artifacts. Accordingly, the Accused Products, through operability with the AV1 Specification, practice Claim Element 5.3. <u>As described with respect to Claim Element 1.3, the decode_frame_wrapup syntax included in the AV1 General tile group OBU syntax is used to call the decode_frame_wrapup function, which triggers the decode frame wrapup process, described in § 7.4, below:</u> S1: § 7.4. Decode frame wrapup process This process is triggered by a call to decode_frame_wrapup from within the syntax tables. At this stage, all the tile level decode has been done, and this process performs any frame level decode that is required. If show_existing_frame is equal to 0, the process first performs any post processing filtering by the following ordered steps: 1. If loop_filter_level[0] is not equal to 0 or loop_filter_level[1] is not equal to 0, the loop filter process specified in section 7.14 is invoked (this process modifies the contents of CurrFrame). 2. The CDEF process specified in section 7.15 is invoked (this process takes CurrFrame and produces CdefFrame). <u>The decode frame wrapup process performs filtering by invoking the ordered steps of (1) the loop filter process and (2) the CDEF process. Accordingly, AV1's loop filter process is the deblocking filter for performing a first pass to reduce blocking artifacts, as is specified in § 7.14, below:</u> S1: § 7.14. Loop filter process S1: § 7.14.1 General Input to this process is the array CurrFrame of reconstructed samples. Output from this process is a modified array CurrFrame containing deblocked samples. The purpose of the loop filter is to eliminate (or at least reduce) visually objectionable artifacts associated with the semi- independence of

CLAIM	PROTOCOL
	the coding of super blocks and their constituent sub-blocks. The loop filter is applied on all vertical boundaries followed by all horizontal boundaries as follows: <pre data-bbox="445 402 1881 885"> for (plane = 0; plane < NumPlanes; plane++) { if (plane == 0 loop_filter_level[1 + plane]) { for (pass = 0; pass < 2; pass++) { rowStep = (plane == 0) ? 1 : (1 << subsampling_y) colStep = (plane == 0) ? 1 : (1 << subsampling_x) for (row = 0; row < MiRows; row += rowStep) for (col = 0; col < MiCols; col += colStep) loop_filter_edge(plane, pass, row, col) } } } </pre> <u>AV1's loop filter is an integral part of the decoding process:</u> Note: The loop filter is an integral part of the decoding process, in that the results of loop filtering are used in the prediction of subsequent frames.
[5.4] and an adaptive sparse de-noising filter for performing a second pass to reduce noise,	The Accused Products either literally and/or under the doctrine of equivalents practice the element “and an adaptive sparse de-noising filter for performing a second pass to reduce noise,” via the AV1 Protocol, as described by at least the following technical specification(s). For example, the Accused Products identified in Appendix 1 practice the AV1 Specification, and, as shown below, the AV1 Specification specifies that a CDEF process is used to perform a second pass to reduce noise and the output of this process is deringed samples. Therefore, the CDEF filter described in the AV1 Specification is an adaptive sparse denoising filter for performing a second pass to reduce noise. Accordingly, the Accused Products, through operability with the AV1 Specification, practice Claim Element 5.4.

CLAIM	PROTOCOL
	<u>The decode frame wrapup process performs filtering by invoking the ordered steps of (1) the loop filter process and (2) the CDEF process. The AV1 CDEF uses a non-linear low-pass filter designed to remove coding artifacts without blurring sharp edges. See, e.g., Midtskogen, et. al, "The AV1 Constrained Directional Enhancement Filter (CDEF)," April 2018, Conference: ICASSP 2018 - 2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), at 3. The filter works by identifying the direction of each block and then adaptively filtering with a high degree of control over the filter strength along the direction and across it. <i>Id.</i>, at 1. The CDEF filter adapts to the sparse nature of underlying video signals by "preventing excessive blurring when the filter is applied across an edge." <i>Id.</i>, at 3. This is done by using the "constraint function $f(d, S, D)$ operating on the difference between the filtered pixel and each of the neighboring pixels. For small differences, $f(d, S, D) = d$, making the filter in (6) behave like a linear filter. When the difference is large, $f(d, S, D) = 0$, which effectively ignores the filter taps" and leads to no filtering. <i>Id.</i> at 3. Accordingly, AV1's CDEF filter process is the adaptive sparse de-noising filter for performing a second pass to reduce noise, as is specified in § 7.15, below:</u> S1: § 7.15. CDEF process Input to this process is the array CurrFrame of reconstructed samples. Output from this process is the array CdefFrame containing deringed samples. The purpose of CDEF is to perform deringing based on the detected direction of blocks. CDEF parameters are stored for each 64 by 64 block of luma samples. The CDEF filter is applied on each 8 by 8 block as follows:

CLAIM	PROTOCOL
	<pre data-bbox="457 289 1900 738"> step4 = Num_4x4_Blocks_Wide[BLOCK_8X8] cdefSize4 = Num_4x4_Blocks_Wide[BLOCK_64X64] cdefMask4 = ~(cdefSize4 - 1) for (r = 0; r < MiRows; r += step4) { for (c = 0; c < MiCols; c += step4) { baseR = r & cdefMask4 baseC = c & cdefMask4 idx = cdef_idx[baseR][baseC] cdef_block(r, c, idx) } } </pre> When the cdef_block function is called, the CDEF block process specified in section 7.15.1 is invoked with r, c, and idx as inputs. <u>The CDEF block process, which in turn invokes the CDEF filter process with the inputs 0, r, c, priStr, secStr, damping, and dir, results from the cdef_block function call, and is described in § 7.15.1, below:</u> S1: § 7.15.1. CDEF block process The inputs to this process are: • variables r and c specifying the location of an 8x8 block in units of 4x4 blocks in the luma plane, • a variable idx specifying which set of CDEF parameters to use, or -1 to signal that no filtering should be applied. The block is first copied to the CdefFrame as follows:

CLAIM	PROTOCOL
	<pre> startY = r * MI_SIZE endY = startY + MI_SIZE * 2 startX = c * MI_SIZE endX = startX + MI_SIZE * 2 for (y = startY; y < endY; y++) { for (x = startX; x < endX; x++) { CdefFrame[0][y][x] = CurrFrame[0][y][x] } } if (NumPlanes > 1) { startY >>= subsampling_y endY >>= subsampling_y startX >>= subsampling_x endX >>= subsampling_x for (y = startY; y < endY; y++) { for (x = startX; x < endX; x++) { CdefFrame[1][y][x] = CurrFrame[1][y][x] CdefFrame[2][y][x] = CurrFrame[2][y][x] } } } </pre> (...) The variable skip is set equal to (Skips[r][c] && Skips[r + 1][c] && Skips[r][c + 1] && Skips[r + 1][c + 1]). If skip is equal to 0, the CDEF direction process specified in section 7.15.2 is invoked with r and c as inputs, and the outputs assigned to variables yDir and var. If skip is equal to 0, the following ordered steps apply: 1. The variable priStr is set equal to cdef_y_pri_strength[idx] << coeffShift.

CLAIM	PROTOCOL
	2. The variable secStr is set equal to $\text{cdef_y_sec_strength}[\text{idx}] \ll \text{coeffShift}$. 3. The variable dir is set equal to $(\text{priStr} == 0) ? 0 : \text{yDir}$. 4. The variable varStr is set equal to $(\text{var} \gg 6) ? \text{Min}(\text{FloorLog2}(\text{var} \gg 6), 12) : 0$. 5. The variable priStr is set equal to $(\text{var} ? (\text{priStr} * (4 + \text{varStr}) + 8) \gg 4 : 0)$. 6. The variable damping is set equal to $\text{CdefDamping} + \text{coeffShift}$. 7. The CDEF filter process specified in section 7.15.3 is invoked with plane equal to 0, r, c, priStr, secStr, damping, and dir as input. 8. If NumPlanes is equal to 1, the process terminates at this point (i.e. filtering is not done for the U and V planes). 9. The variable priStr is set equal to $\text{cdef_uv_pri_strength}[\text{idx}] \ll \text{coeffShift}$. 10. The variable secStr is set equal to $\text{cdef_uv_sec_strength}[\text{idx}] \ll \text{coeffShift}$. 11. The variable dir is set equal to $(\text{priStr} == 0) ? 0 : \text{Cdef_Uv_Dir}[\text{subsampling_x}][\text{subsampling_y}][\text{yDir}]$. 12. The variable damping is set equal to $\text{CdefDamping} + \text{coeffShift} - 1$. 13. The CDEF filter process specified in section 7.15.3 is invoked with plane equal to 1, r, c, priStr, secStr, damping, and dir as input.

CLAIM	PROTOCOL
	14. The CDEF filter process specified in section 7.15.3 is invoked with plane equal to 2, r, c, priStr, secStr, damping, and dir as input. Cdef_Uv_Dir is a constant lookup table defined as: <pre data-bbox="464 462 884 667">Cdef_Uv_Dir[2][2][8] = { { {0, 1, 2, 3, 4, 5, 6, 7}, {1, 2, 2, 2, 3, 4, 6, 0} }, { {7, 0, 2, 4, 5, 6, 6, 6}, {0, 1, 2, 3, 4, 5, 6, 7} } }</pre> <u>The CDEF params syntax are described in § 5.9.19, below:</u> S1: § 5.9.19. CDEF params syntax

CLAIM	PROTOCOL																																																								
	<table> <tr> <th></th><th>Type</th></tr> <tr> <td>cdef_params() {</td><td></td></tr> <tr> <td> if (CodedLossless allow_intrabc </td><td></td></tr> <tr> <td> !enable_cdef) {</td><td></td></tr> <tr> <td> cdef_bits = 0</td><td></td></tr> <tr> <td> cdef_y_pri_strength[0] = 0</td><td></td></tr> <tr> <td> cdef_y_sec_strength[0] = 0</td><td></td></tr> <tr> <td> cdef_uv_pri_strength[0] = 0</td><td></td></tr> <tr> <td> cdef_uv_sec_strength[0] = 0</td><td></td></tr> <tr> <td> CdefDamping = 3</td><td></td></tr> <tr> <td> return</td><td></td></tr> <tr> <td> }</td><td></td></tr> <tr> <td> cdef_damping_minus_3</td><td>f(2)</td></tr> <tr> <td> CdefDamping = cdef_damping_minus_3 + 3</td><td></td></tr> <tr> <td> cdef_bits</td><td>f(2)</td></tr> <tr> <td> for (i = 0; i < (1 << cdef_bits); i++) {</td><td></td></tr> <tr> <td> cdef_y_pri_strength[i]</td><td>f(4)</td></tr> <tr> <td> cdef_y_sec_strength[i]</td><td>f(2)</td></tr> <tr> <td> if (cdef_y_sec_strength[i] == 3)</td><td></td></tr> <tr> <td> cdef_y_sec_strength[i] += 1</td><td></td></tr> <tr> <td> if (NumPlanes > 1) {</td><td></td></tr> <tr> <td> cdef_uv_pri_strength[i]</td><td>f(4)</td></tr> <tr> <td> cdef_uv_sec_strength[i]</td><td>f(2)</td></tr> <tr> <td> if (cdef_uv_sec_strength[i] == 3)</td><td></td></tr> <tr> <td> cdef_uv_sec_strength[i] += 1</td><td></td></tr> <tr> <td> }</td><td></td></tr> <tr> <td> }</td><td></td></tr> <tr> <td>}</td><td></td></tr> </table> <u>The semantics of the CDEF params syntax are described in § 6.10.14:</u> S1: § 6.10.14. CDEF params semantics cdef_damping_minus_3 controls the amount of damping in the deringing filter.		Type	cdef_params() {		if (CodedLossless allow_intrabc		!enable_cdef) {		cdef_bits = 0		cdef_y_pri_strength[0] = 0		cdef_y_sec_strength[0] = 0		cdef_uv_pri_strength[0] = 0		cdef_uv_sec_strength[0] = 0		CdefDamping = 3		return		}		cdef_damping_minus_3	f(2)	CdefDamping = cdef_damping_minus_3 + 3		cdef_bits	f(2)	for (i = 0; i < (1 << cdef_bits); i++) {		cdef_y_pri_strength[i]	f(4)	cdef_y_sec_strength[i]	f(2)	if (cdef_y_sec_strength[i] == 3)		cdef_y_sec_strength[i] += 1		if (NumPlanes > 1) {		cdef_uv_pri_strength[i]	f(4)	cdef_uv_sec_strength[i]	f(2)	if (cdef_uv_sec_strength[i] == 3)		cdef_uv_sec_strength[i] += 1		}		}		}	
	Type																																																								
cdef_params() {																																																									
if (CodedLossless allow_intrabc																																																									
!enable_cdef) {																																																									
cdef_bits = 0																																																									
cdef_y_pri_strength[0] = 0																																																									
cdef_y_sec_strength[0] = 0																																																									
cdef_uv_pri_strength[0] = 0																																																									
cdef_uv_sec_strength[0] = 0																																																									
CdefDamping = 3																																																									
return																																																									
}																																																									
cdef_damping_minus_3	f(2)																																																								
CdefDamping = cdef_damping_minus_3 + 3																																																									
cdef_bits	f(2)																																																								
for (i = 0; i < (1 << cdef_bits); i++) {																																																									
cdef_y_pri_strength[i]	f(4)																																																								
cdef_y_sec_strength[i]	f(2)																																																								
if (cdef_y_sec_strength[i] == 3)																																																									
cdef_y_sec_strength[i] += 1																																																									
if (NumPlanes > 1) {																																																									
cdef_uv_pri_strength[i]	f(4)																																																								
cdef_uv_sec_strength[i]	f(2)																																																								
if (cdef_uv_sec_strength[i] == 3)																																																									
cdef_uv_sec_strength[i] += 1																																																									
}																																																									
}																																																									
}																																																									

CLAIM	PROTOCOL
	Cdef_bits specifies the number of bits needed to specify which CDEF filter to apply. Cdef_y_pri_strength and cdef_uv_pri_strength specify the strength of the primary filter. Cdef_y_sec_strength and cdef_uv_sec_strength specify the strength of the secondary filter. <u>As described above in § 7.15.1, the ordered steps of the CDEF filter process utilize the outputs yDir and var of the CDEF direction process, described below in § 7.15.2:</u> S1: § 7.15.2 CDEF direction process The inputs to this process are variables r and c specifying the location of an 8x8 block in units of 4x4 blocks in the luma plane. The outputs of this process are: • a variable yDir containing the direction of this block, • a variable var containing the variance for this block. This block uses luma samples to measure the direction and variance of a block. The process is specified as:

CLAIM	PROTOCOL
	<pre> for (i = 0; i < 8; i++) { cost[i] = 0 for (j = 0; j < 15; j++) partial[i][j] = 0 } bestCost = 0 yDir = 0 x0 = c << MI_SIZE_LOG2 y0 = r << MI_SIZE_LOG2 for (i = 0; i < 8; i++) { for (j = 0; j < 8; j++) { x = (CurrFrame[0][y0 + i][x0 + j] >> (BitDepth - 8)) - 128 partial[0][i + j] += x partial[1][i + j / 2] += x partial[2][i] += x partial[3][3 + i - j / 2] += x partial[4][7 + i - j] += x partial[5][3 - i / 2 + j] += x partial[6][j] += x partial[7][i / 2 + j] += x } } for (i = 0; i < 8; i++) { cost[2] += partial[2][i] * partial[2][i] cost[6] += partial[6][i] * partial[6][i] } cost[2] *= Div_Table[8] cost[6] *= Div_Table[8] for (i = 0; i < 7; i++) { cost[0] += (partial[0][i] * partial[0][i] + partial[0][14 - i] * partial[0][14 - i]) * Div_Table[i + 1] </pre>

CLAIM	PROTOCOL
	<pre> cost[4] += (partial[4][i] * partial[4][i] + partial[4][14 - i] * partial[4][14 - i]) * Div_Table[i + 1] } cost[0] += partial[0][7] * partial[0][7] * Div_Table[8] cost[4] += partial[4][7] * partial[4][7] * Div_Table[8] for (i = 1; i < 8; i += 2) { for (j = 0; j < 4 + 1; j++) { cost[i] += partial[i][3 + j] * partial[i][3 + j] } cost[i] *= Div_Table[8] for (j = 0; j < 4 - 1; j++) { cost[i] += (partial[i][j] * partial[i][j] + partial[i][10 - j] * partial[i][10 - j]) * Div_Table[2 * j + 2] } } for (i = 0; i < 8; i++) { if (cost[i] > bestCost) { bestCost = cost[i] yDir = i } } var = (bestCost - cost[(yDir + 4) & 7]) >> 10 </pre> where the Div_Table is a constant lookup table specified as:

CLAIM	PROTOCOL
	<pre>Div_Table[9] = { 0, 840, 420, 280, 210, 168, 140, 120, 105 }</pre> The adaptive sparse CDEF filter process invoked by the ordered steps in the CDEF block process above is applied as a second pass to reduce quantization noise by removing coding artifacts, and is described in § 7.15.3, below. <i>See, e.g.</i>, Midstskogen, et. al, “The AV1 Constrained Directional Enhancement Filter (CDEF).” April 2018, Conference: ICASSP 2018 - 2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), at 3 (“CDEF uses a non-linear low-pass filter designed to remove coding artifacts without blurring sharp edges. It achieves this by selecting filter tap locations based on the identified direction, but also by preventing excessive blurring when the filter is applied across an edge.”); <i>see also id.</i>, 1 (“The main goal of the in-loop constrained directional enhancement filter (CDEF) is to filter out coding artifacts while retaining the details of the image The amount of ringing artifacts in a coded image tends to be roughly proportional to the quantization step size. The amount of detail is a property of the input image, but the smallest detail actually retained in the quantized image tends to also be proportional to the quantization step size. For a given quantization step size, the amplitude of the ringing is generally less than the amplitude of the details. CDEF works by identifying the direction [4] of each block (Sec. 2) and then adaptively filtering along the identified direction (Sec. 3)”). S1: §7.15.3. CDEF filter process The inputs to this process are: • a variable plane specifying which plane is being predicted, • variables r and c specifying the location of an 8x8 block in units of 4x4 blocks in the luma plane, • a variable priStr specifying the primary filter strength, • a variable secStr specifying the secondary filter strength, • a variable damping specifying a shift used for damping, • a variable dir specifying the detected direction of the block. The process modifies samples in CdefFrame based on filtering samples from CurrFrame.

Exhibit 5 to InterDigital's Preliminary Infringement Contentions

CLAIM	PROTOCOL
	MiColStart, MiRowStart, MiColEnd, MiRowEnd are set equal to the values they had when the syntax element MiSizes[r][c] was written. (...) The filtering is applied as follows: (...)

```

subX = (plane > 0) ? subsampling_x : 0
subY = (plane > 0) ? subsampling_y : 0
x0 = (c * MI_SIZE) >> subX
y0 = (r * MI_SIZE) >> subY
w = 8 >> subX
h = 8 >> subY
for ( i = 0; i < h; i++ ) {
    for ( j = 0; j < w; j++ ) {
        sum = 0
        x = CurrFrame[plane][y0 + i][x0 + j]
        max = x
        min = x
        for ( k = 0; k < 2; k++ ) {
            for ( sign = -1; sign <= 1; sign += 2 ) {
                p = cdef_get_at(plane, x0, y0, i, j, dir, k, sign, subX, subY)
                if ( CdefAvailable ) {
                    sum += Cdef_Pri_Taps[(priStr >> coeffShift) & 1][k] * constrain(p - x, priStr,
damping)

                    max = Max(p, max)
                    min = Min(p, min)
                }
                for ( dirOff = -2; dirOff <= 2; dirOff += 4 ) {
                    s = cdef_get_at(plane, x0, y0, i, j, (dir + dirOff) & 7, k, sign, subX, subY)
                    if ( CdefAvailable ) {
                        sum += Cdef_Sec_Taps[(priStr >> coeffShift) & 1][k] * constrain(s - x,
secStr, damping)

                        max = Max(s, max)
                        min = Min(s, min)
                    }
                }
            }
        }
        CdefFrame[plane][y0 + i][x0 + j] = Clip3(min, max, x + ((8 + sum - (sum < 0)) >> 4) )
    }
}

```

CLAIM	PROTOCOL
	where Cdef_Pri_Taps and Cdef_Sec_Taps are constant lookup tables specified as: <pre> Cdef_Pri_Taps[2][2] = { { 4, 2 }, { 3, 3 } } Cdef_Sec_Taps[2][2] = { { 2, 1 }, { 2, 1 } } </pre> constrain is specified as: <pre> constrain(diff, threshold, damping) { if (!threshold) return 0 dampingAdj = Max(0, damping - FloorLog2(threshold)) sign = (diff < 0) ? -1 : 1 return sign * Clip3(0, Abs(diff), threshold - (Abs(diff) >> dampingAdj)) } </pre> (...) where Cdef_Directions is a constant lookup table defined as:

CLAIM	PROTOCOL
	<pre> Cdef_Directions[8][2][2] = { { { -1, 1 }, { -2, 2 } }, { { 0, 1 }, { -1, 2 } }, { { 0, 1 }, { 0, 2 } }, { { 0, 1 }, { 1, 2 } }, { { 1, 1 }, { 2, 2 } }, { { 1, 0 }, { 2, 1 } }, { { 1, 0 }, { 2, 0 } }, { { 1, 0 }, { 2, -1 } } } </pre>
[5.5] wherein the adaptive sparse de-noising filter is selectively enabled or disabled at a given level, the given level being at least a macroblock level, a slice level, a picture level or a sequence level.	The Accused Products either literally and/or under the doctrine of equivalents practice the element “wherein the adaptive sparse de-noising filter is selectively enabled or disabled at a given level, the given level being at least a macroblock level, a slice level, a picture level or a sequence level” via the AV1 Protocol, as described by at least the following technical specification(s). For example, the Accused Products identified in Appendix 1 practice the AV1 Specification, and, as shown below, the AV1 Specification specifies that the CDEF adaptive sparse de-noising filter is selectively enabled or disabled at a given level. The AV 1 Specification further describes that the CDEF filter is enabled or disabled at least at a macroblock level, a slice level, a picture level or a sequence level. Accordingly, the Accused Products, through operability with the AV1 Specification, practice Claim Element 5.5. <u>The AV1 bitstream uses the general OBU syntax as described in § 5.3, which includes the sequence_header_obu function.</u> S1: § 5.3. OBU syntax S1: § 5.3.1. General OBU syntax

CLAIM	PROTOCOL																																												
	<table> <tr> <td>open_bitstream_unit(sz) {</td><td>Type</td></tr> <tr> <td> obu_header()</td><td></td></tr> <tr> <td> if (obu_has_size_field) {</td><td></td></tr> <tr> <td> obu_size</td><td>leb128()</td></tr> <tr> <td> } else {</td><td></td></tr> <tr> <td> obu_size = sz - 1 - obu_extension_flag</td><td></td></tr> <tr> <td> }</td><td></td></tr> <tr> <td> startPosition = get_position()</td><td></td></tr> <tr> <td> if (obu_type != OBU_SEQUENCE_HEADER &&</td><td></td></tr> <tr> <td> obu_type != OBU_TEMPORAL_DELIMITER &&</td><td></td></tr> <tr> <td> OperatingPointIdc != 0 &&</td><td></td></tr> <tr> <td> obu_extension_flag == 1)</td><td></td></tr> <tr> <td> {</td><td></td></tr> <tr> <td> inTemporalLayer = (OperatingPointIdc >> temporal_id) & 1</td><td></td></tr> <tr> <td> inSpatialLayer = (OperatingPointIdc >> (spatial_id + 8)) & 1</td><td></td></tr> <tr> <td> if (!inTemporalLayer ! inSpatialLayer) {</td><td></td></tr> <tr> <td> drop_obu()</td><td></td></tr> <tr> <td> return</td><td></td></tr> <tr> <td> }</td><td></td></tr> <tr> <td> }</td><td></td></tr> <tr> <td> if (obu_type == OBU_SEQUENCE_HEADER)</td><td></td></tr> <tr> <td> sequence_header_obu()</td><td></td></tr> </table> The syntax structure for the AV1 sequence_header_obu function includes “enable_cdef,” as is described in § 5.5, below. S1: § 5.5. Sequence header OBU syntax S1: § 5.5.1. General	open_bitstream_unit(sz) {	Type	obu_header()		if (obu_has_size_field) {		obu_size	leb128()	} else {		obu_size = sz - 1 - obu_extension_flag		}		startPosition = get_position()		if (obu_type != OBU_SEQUENCE_HEADER &&		obu_type != OBU_TEMPORAL_DELIMITER &&		OperatingPointIdc != 0 &&		obu_extension_flag == 1)		{		inTemporalLayer = (OperatingPointIdc >> temporal_id) & 1		inSpatialLayer = (OperatingPointIdc >> (spatial_id + 8)) & 1		if (!inTemporalLayer ! inSpatialLayer) {		drop_obu()		return		}		}		if (obu_type == OBU_SEQUENCE_HEADER)		sequence_header_obu()	
open_bitstream_unit(sz) {	Type																																												
obu_header()																																													
if (obu_has_size_field) {																																													
obu_size	leb128()																																												
} else {																																													
obu_size = sz - 1 - obu_extension_flag																																													
}																																													
startPosition = get_position()																																													
if (obu_type != OBU_SEQUENCE_HEADER &&																																													
obu_type != OBU_TEMPORAL_DELIMITER &&																																													
OperatingPointIdc != 0 &&																																													
obu_extension_flag == 1)																																													
{																																													
inTemporalLayer = (OperatingPointIdc >> temporal_id) & 1																																													
inSpatialLayer = (OperatingPointIdc >> (spatial_id + 8)) & 1																																													
if (!inTemporalLayer ! inSpatialLayer) {																																													
drop_obu()																																													
return																																													
}																																													
}																																													
if (obu_type == OBU_SEQUENCE_HEADER)																																													
sequence_header_obu()																																													

CLAIM	PROTOCOL																												
	<table border="1"> <thead> <tr> <th></th><th>Type</th></tr> </thead> <tbody> <tr> <td>sequence_header_obu() {</td><td></td></tr> <tr> <td> seq_profile</td><td>f(3)</td></tr> <tr> <td> still_picture</td><td>f(1)</td></tr> <tr> <td> reduced_still_picture_header</td><td>f(1)</td></tr> <tr> <td> if (reduced_still_picture_header) {</td><td></td></tr> <tr> <td> (...)</td><td></td></tr> <tr> <td> }</td><td></td></tr> <tr> <td> enable_superres</td><td>f(1)</td></tr> <tr> <td> enable_cdef</td><td>f(1)</td></tr> <tr> <td> enable_restoration</td><td>f(1)</td></tr> <tr> <td> color_config()</td><td></td></tr> <tr> <td> film_grain_params_present</td><td>f(1)</td></tr> <tr> <td>}</td><td></td></tr> </tbody> </table> <u>The semantics of the sequence_header_obu syntax, including “enable_cdef”, are described in § 6.4, below:</u> S1: § 6.4. Sequence header OBU semantics S1: § 6.4.1. General sequence header OBU semantics (...) enable_cdef equal to 1 specifies that cdef filtering may be enabled. enable_cdef equal to 0 specifies that cdef filtering is disabled. Note: It is allowed to set enable_cdef equal to 1 even when cdef filtering is not used on any frame in the coded video sequence. <u>As part of the tile group OBU syntax, § 5.11, the decode block syntax is provided, which includes the mode_info function</u> S1: § 5.11.5. Decode block syntax		Type	sequence_header_obu() {		seq_profile	f(3)	still_picture	f(1)	reduced_still_picture_header	f(1)	if (reduced_still_picture_header) {		(...)		}		enable_superres	f(1)	enable_cdef	f(1)	enable_restoration	f(1)	color_config()		film_grain_params_present	f(1)	}	
	Type																												
sequence_header_obu() {																													
seq_profile	f(3)																												
still_picture	f(1)																												
reduced_still_picture_header	f(1)																												
if (reduced_still_picture_header) {																													
(...)																													
}																													
enable_superres	f(1)																												
enable_cdef	f(1)																												
enable_restoration	f(1)																												
color_config()																													
film_grain_params_present	f(1)																												
}																													

		Type
	decode_block(r, c, subSize) {	
	MiRow = r	
	MiCol = c	
	MiSize = subSize	
	bw4 = Num_4x4_Blocks_Wide[subSize]	
	bh4 = Num_4x4_Blocks_High[subSize]	
	if (bh4 == 1 && subsampling_y && (MiRow & 1) == 0)	
	HasChroma = 0	
	else if (bw4 == 1 && subsampling_x && (MiCol & 1) == 0)	
	HasChroma = 0	
	else	
	HasChroma = NumPlanes > 1	
	AvailU = is_inside(r - 1, c)	
	AvailL = is_inside(r, c - 1)	
	AvailUChroma = AvailU	
	AvailLChroma = AvailL	
	if (HasChroma) {	
	if (subsampling_y && bh4 == 1)	
	AvailUChroma = is_inside(r - 2, c)	
	if (subsampling_x && bw4 == 1)	
	AvailLChroma = is_inside(r, c - 2)	
	} else {	
	AvailUChroma = 0	
	AvailLChroma = 0	
	}	
	mode_info()	
	palette_tokens()	
	read_block_tx_size()	
	if (skip)	
	reset_block_context(bw4, bh4)	
	isCompound = RefFrame[1] > INTRA_FRAME	

	for (y = 0; y < bh4; y++) {	
	for (x = 0; x < bw4; x++) {	
	YModes [r + y][c + x] = YMode	
	if (RefFrame[0] == INTRA_FRAME && HasChroma)	
	UVModes [r + y][c + x] = UVMode	
	for (refList = 0; refList < 2; refList++)	
	RefFrames[r + y][c + x][refList] = RefFrame[refList]	
	if (is_inter) {	
	if (!use_intrabc) {	
	CompGroupIdxs[r + y][c + x] = comp_group_idx	
	CompoundIdxs[r + y][c + x] = compound_idx	
	}	
	for (dir = 0; dir < 2; dir++) {	
	InterpFilters[r + y][c + x][dir] = interp_filter[dir]	
	}	
	for (refList = 0; refList < 1 + isCompound; refList++) {	
	Mvs[r + y][c + x][refList] = Mv[refList]	
	}	
	}	
	}	
	compute_prediction()	
	residual()	
	for (y = 0; y < bh4; y++) {	
	for (x = 0; x < bw4; x++) {	
	IsInters[r + y][c + x] = is_inter	
	SkipModes[r + y][c + x] = skip_mode	
	Skips[r + y][c + x] = skip	
	TxSizes[r + y][c + x] = TxSize	
	MiSizes[r + y][c + x] = MiSize	
	SegmentIds[r + y][c + x] = segment_id	
	PaletteSizes[0][r + y][c + x] = PaletteSizeY	

CLAIM	PROTOCOL																																
	<table> <tr><td>PaletteSizes[1][r + y][c + x] = PaletteSizeUV</td><td></td></tr> <tr><td>for (i = 0; i < PaletteSizeY; i++)</td><td></td></tr> <tr><td> PaletteColors[0][r + y][c + x][i] = palette_colors_y[i]</td><td></td></tr> <tr><td>for (i = 0; i < PaletteSizeUV; i++)</td><td></td></tr> <tr><td> PaletteColors[1][r + y][c + x][i] = palette_colors_u[i]</td><td></td></tr> <tr><td>for (i = 0; i < FRAME_LF_COUNT; i++)</td><td></td></tr> <tr><td> DeltaLFs[r + y][c + x][i] = DeltaLF[i]</td><td></td></tr> <tr><td> }</td><td></td></tr> <tr><td> }</td><td></td></tr> <tr><td>}</td><td></td></tr> </table> The mode_info function called in the above decode block is described in § 5.11.6, below: S1: §5.11.6. Mode info syntax <table> <tr><td>mode_info() {</td><td>Type</td></tr> <tr><td> if (FrameIsIntra)</td><td></td></tr> <tr><td> intra_frame_mode_info()</td><td></td></tr> <tr><td> else</td><td></td></tr> <tr><td> inter_frame_mode_info()</td><td></td></tr> <tr><td> }</td><td></td></tr> </table> When mode_info is not FrameIsIntra, the inter_frame_mode_info function is called, as described in § 5.11.P8, below: S1: § 5.11.P8. Inter frame mode info syntax	PaletteSizes[1][r + y][c + x] = PaletteSizeUV		for (i = 0; i < PaletteSizeY; i++)		PaletteColors[0][r + y][c + x][i] = palette_colors_y[i]		for (i = 0; i < PaletteSizeUV; i++)		PaletteColors[1][r + y][c + x][i] = palette_colors_u[i]		for (i = 0; i < FRAME_LF_COUNT; i++)		DeltaLFs[r + y][c + x][i] = DeltaLF[i]		}		}		}		mode_info() {	Type	if (FrameIsIntra)		intra_frame_mode_info()		else		inter_frame_mode_info()		}	
PaletteSizes[1][r + y][c + x] = PaletteSizeUV																																	
for (i = 0; i < PaletteSizeY; i++)																																	
PaletteColors[0][r + y][c + x][i] = palette_colors_y[i]																																	
for (i = 0; i < PaletteSizeUV; i++)																																	
PaletteColors[1][r + y][c + x][i] = palette_colors_u[i]																																	
for (i = 0; i < FRAME_LF_COUNT; i++)																																	
DeltaLFs[r + y][c + x][i] = DeltaLF[i]																																	
}																																	
}																																	
}																																	
mode_info() {	Type																																
if (FrameIsIntra)																																	
intra_frame_mode_info()																																	
else																																	
inter_frame_mode_info()																																	
}																																	

Exhibit 5 to InterDigital's Preliminary Infringement Contentions

CLAIM	PROTOCOL																																																														
	<table> <tr> <th></th><th>Type</th></tr> <tr> <td>inter_frame_mode_info() {</td><td></td></tr> <tr> <td> use_intrabc = 0</td><td></td></tr> <tr> <td> LeftRefFrame[0] = AvailL ? RefFrames[MiRow][MiCol-1][0] : INTRA_FRAME</td><td></td></tr> <tr> <td> AboveRefFrame[0] = AvailU ? RefFrames[MiRow-1][MiCol][0] : INTRA_FRAME</td><td></td></tr> <tr> <td> LeftRefFrame[1] = AvailL ? RefFrames[MiRow][MiCol-1][1] : NONE</td><td></td></tr> <tr> <td> AboveRefFrame[1] = AvailU ? RefFrames[MiRow-1][MiCol][1] : NONE</td><td></td></tr> <tr> <td> LeftIntra = LeftRefFrame[0] <= INTRA_FRAME</td><td></td></tr> <tr> <td> AboveIntra = AboveRefFrame[0] <= INTRA_FRAME</td><td></td></tr> <tr> <td> LeftSingle = LeftRefFrame[1] <= INTRA_FRAME</td><td></td></tr> <tr> <td> AboveSingle = AboveRefFrame[1] <= INTRA_FRAME</td><td></td></tr> <tr> <td> skip = 0</td><td></td></tr> <tr> <td> inter_segment_id(1)</td><td></td></tr> <tr> <td> read_skip_mode()</td><td></td></tr> <tr> <td> if (skip_mode)</td><td></td></tr> <tr> <td> skip = 1</td><td></td></tr> <tr> <td> else</td><td></td></tr> <tr> <td> read_skip()</td><td></td></tr> <tr> <td> if (!SegIdPreSkip)</td><td></td></tr> <tr> <td> inter_segment_id(0)</td><td></td></tr> <tr> <td> Lossless = LosslessArray[segment_id]</td><td></td></tr> <tr> <td> read_cdef()</td><td></td></tr> <tr> <td> read_delta_qindex()</td><td></td></tr> <tr> <td> read_delta_lf()</td><td></td></tr> <tr> <td> ReadDeltas = 0</td><td></td></tr> <tr> <td> read_is_inter()</td><td></td></tr> <tr> <td> if (is_inter)</td><td></td></tr> <tr> <td> inter_block_mode_info()</td><td></td></tr> <tr> <td> else</td><td></td></tr> <tr> <td> intra_block_mode_info()</td><td></td></tr> <tr> <td> }</td><td></td></tr> </table>		Type	inter_frame_mode_info() {		use_intrabc = 0		LeftRefFrame[0] = AvailL ? RefFrames[MiRow][MiCol-1][0] : INTRA_FRAME		AboveRefFrame[0] = AvailU ? RefFrames[MiRow-1][MiCol][0] : INTRA_FRAME		LeftRefFrame[1] = AvailL ? RefFrames[MiRow][MiCol-1][1] : NONE		AboveRefFrame[1] = AvailU ? RefFrames[MiRow-1][MiCol][1] : NONE		LeftIntra = LeftRefFrame[0] <= INTRA_FRAME		AboveIntra = AboveRefFrame[0] <= INTRA_FRAME		LeftSingle = LeftRefFrame[1] <= INTRA_FRAME		AboveSingle = AboveRefFrame[1] <= INTRA_FRAME		skip = 0		inter_segment_id(1)		read_skip_mode()		if (skip_mode)		skip = 1		else		read_skip()		if (!SegIdPreSkip)		inter_segment_id(0)		Lossless = LosslessArray[segment_id]		read_cdef()		read_delta_qindex()		read_delta_lf()		ReadDeltas = 0		read_is_inter()		if (is_inter)		inter_block_mode_info()		else		intra_block_mode_info()		}	
	Type																																																														
inter_frame_mode_info() {																																																															
use_intrabc = 0																																																															
LeftRefFrame[0] = AvailL ? RefFrames[MiRow][MiCol-1][0] : INTRA_FRAME																																																															
AboveRefFrame[0] = AvailU ? RefFrames[MiRow-1][MiCol][0] : INTRA_FRAME																																																															
LeftRefFrame[1] = AvailL ? RefFrames[MiRow][MiCol-1][1] : NONE																																																															
AboveRefFrame[1] = AvailU ? RefFrames[MiRow-1][MiCol][1] : NONE																																																															
LeftIntra = LeftRefFrame[0] <= INTRA_FRAME																																																															
AboveIntra = AboveRefFrame[0] <= INTRA_FRAME																																																															
LeftSingle = LeftRefFrame[1] <= INTRA_FRAME																																																															
AboveSingle = AboveRefFrame[1] <= INTRA_FRAME																																																															
skip = 0																																																															
inter_segment_id(1)																																																															
read_skip_mode()																																																															
if (skip_mode)																																																															
skip = 1																																																															
else																																																															
read_skip()																																																															
if (!SegIdPreSkip)																																																															
inter_segment_id(0)																																																															
Lossless = LosslessArray[segment_id]																																																															
read_cdef()																																																															
read_delta_qindex()																																																															
read_delta_lf()																																																															
ReadDeltas = 0																																																															
read_is_inter()																																																															
if (is_inter)																																																															
inter_block_mode_info()																																																															
else																																																															
intra_block_mode_info()																																																															
}																																																															

CLAIM	PROTOCOL																																										
	<u>The read_cdef function called in the above inter frame mode selectively enables or disables the CDEF filter using enable_cdef and cdef_idx, as described in § 5.11.56, below:</u> S1: § 5.11.56. Read CDEF syntax <table data-bbox="438 448 1885 1149"> <tr> <td data-bbox="451 456 1598 488">read_cdef() {</td><td data-bbox="1598 456 1885 488">Type</td></tr> <tr> <td data-bbox="451 488 1598 521"> if (skip CodedLossless !enable_cdef allow_intrabc) {</td><td data-bbox="1598 488 1885 521"></td></tr> <tr> <td data-bbox="451 521 1598 553"> return</td><td data-bbox="1598 521 1885 553"></td></tr> <tr> <td data-bbox="451 553 1598 586"> }</td><td data-bbox="1598 553 1885 586"></td></tr> <tr> <td data-bbox="451 586 1598 618"> cdefSize4 = Num_4x4_Blocks_Wide[BLOCK_64X64]</td><td data-bbox="1598 586 1885 618"></td></tr> <tr> <td data-bbox="451 618 1598 651"> cdefMask4 = ~(cdefSize4 - 1)</td><td data-bbox="1598 618 1885 651"></td></tr> <tr> <td data-bbox="451 651 1598 683"> r = MiRow & cdefMask4</td><td data-bbox="1598 651 1885 683"></td></tr> <tr> <td data-bbox="451 683 1598 716"> c = MiCol & cdefMask4</td><td data-bbox="1598 683 1885 716"></td></tr> <tr> <td data-bbox="451 716 1598 748"> if (cdef_idx[r][c] == -1) {</td><td data-bbox="1598 716 1885 748"></td></tr> <tr> <td data-bbox="451 748 1598 781"> cdef_idx[r][c]</td><td data-bbox="1598 748 1885 781">L(cdef_bits)</td></tr> <tr> <td data-bbox="451 781 1598 813"> w4 = Num_4x4_Blocks_Wide[MiSize]</td><td data-bbox="1598 781 1885 813"></td></tr> <tr> <td data-bbox="451 813 1598 846"> h4 = Num_4x4_Blocks_High[MiSize]</td><td data-bbox="1598 813 1885 846"></td></tr> <tr> <td data-bbox="451 846 1598 878"> for (i = r; i < r + h4 ; i += cdefSize4) {</td><td data-bbox="1598 846 1885 878"></td></tr> <tr> <td data-bbox="451 878 1598 911"> for (j = c; j < c + w4 ; j += cdefSize4) {</td><td data-bbox="1598 878 1885 911"></td></tr> <tr> <td data-bbox="451 911 1598 943"> cdef_idx[i][j] = cdef_idx[r][c]</td><td data-bbox="1598 911 1885 943"></td></tr> <tr> <td data-bbox="451 943 1598 976"> }</td><td data-bbox="1598 943 1885 976"></td></tr> <tr> <td data-bbox="451 976 1598 1008"> }</td><td data-bbox="1598 976 1885 1008"></td></tr> <tr> <td data-bbox="451 1008 1598 1040"> }</td><td data-bbox="1598 1008 1885 1040"></td></tr> <tr> <td data-bbox="451 1040 1598 1073">}</td><td data-bbox="1598 1040 1885 1073"></td></tr> <tr> <td data-bbox="451 1073 1598 1105"></td><td data-bbox="1598 1073 1885 1105"></td></tr> <tr> <td data-bbox="451 1105 1598 1138"></td><td data-bbox="1598 1105 1885 1138"></td></tr> </table> <u>The semantics for the Read CDEF function is described in § 6.10.40, below:</u> S1: § 6.10.40. Read CDEF semantics cdef_idx specifies which CDEF filtering parameters should be used for a particular 64 by 64 block. A value of -1 means that CDEF is disabled for that block.	read_cdef() {	Type	if (skip CodedLossless !enable_cdef allow_intrabc) {		return		}		cdefSize4 = Num_4x4_Blocks_Wide[BLOCK_64X64]		cdefMask4 = ~(cdefSize4 - 1)		r = MiRow & cdefMask4		c = MiCol & cdefMask4		if (cdef_idx[r][c] == -1) {		cdef_idx[r][c]	L(cdef_bits)	w4 = Num_4x4_Blocks_Wide[MiSize]		h4 = Num_4x4_Blocks_High[MiSize]		for (i = r; i < r + h4 ; i += cdefSize4) {		for (j = c; j < c + w4 ; j += cdefSize4) {		cdef_idx[i][j] = cdef_idx[r][c]		}		}		}		}					
read_cdef() {	Type																																										
if (skip CodedLossless !enable_cdef allow_intrabc) {																																											
return																																											
}																																											
cdefSize4 = Num_4x4_Blocks_Wide[BLOCK_64X64]																																											
cdefMask4 = ~(cdefSize4 - 1)																																											
r = MiRow & cdefMask4																																											
c = MiCol & cdefMask4																																											
if (cdef_idx[r][c] == -1) {																																											
cdef_idx[r][c]	L(cdef_bits)																																										
w4 = Num_4x4_Blocks_Wide[MiSize]																																											
h4 = Num_4x4_Blocks_High[MiSize]																																											
for (i = r; i < r + h4 ; i += cdefSize4) {																																											
for (j = c; j < c + w4 ; j += cdefSize4) {																																											
cdef_idx[i][j] = cdef_idx[r][c]																																											
}																																											
}																																											
}																																											
}																																											

CLAIM	PROTOCOL
	<u>The CDEF process, including input cdef_idx, is then described in § 7.15, below:</u> S1: § 7.15. CDEF process Input to this process is the array CurrFrame of reconstructed samples. Output from this process is the array CdefFrame containing 70eranged samples. The purpose of CDEF is to perform deringing based on the detected direction of blocks. CDEF parameters are stored for each 64 by 64 block of luma samples. The CDEF filter is applied on each 8 by 8 block as follows: <pre> step4 = Num_4x4_Blocks_Wide[BLOCK_8X8] cdefSize4 = Num_4x4_Blocks_Wide[BLOCK_64X64] cdefMask4 = ~(cdefSize4 - 1) for (r = 0; r < MiRows; r += step4) { for (c = 0; c < MiCols; c += step4) { baseR = r & cdefMask4 baseC = c & cdefMask4 idx = cdef_idx[baseR][baseC] cdef_block(r, c, idx) } } </pre> When the cdef_block function is called, the CDEF block process specified in section 7.15.1 is invoked with r, c, and idx as inputs. S1: § 7.15.1. CDEF block process The inputs to this process are:

CLAIM	PROTOCOL
	• variables r and c specifying the location of an 8x8 block in units of 4x4 blocks in the luma plane, • a variable idx specifying which set of CDEF parameters to use, or -1 to signal that no filtering should be applied.
6. The apparatus of claim 5, wherein at least one of the at least two filters is used to enforce at least one compression constraint.	Given the Accused Products decode according to AV1, upon information and belief and subject to further discovery, the Accused Products likely either literally and/or under the doctrine of equivalents practice the claim “wherein at least one of the at least two filters is used to enforce at least one compression constraint.” <i>See Claim 5, which describes the underlying independent claim.</i>
7. The apparatus of claim 5, wherein at least one of the at least two filters is adaptively enabled or disabled depending upon at least data characteristics corresponding to the image region or data statistics corresponding to the image region.	The Accused Products either literally and/or under the doctrine of equivalents practice the claim “at least one of the at least two filters is adaptively enabled or disabled depending upon at least data characteristics corresponding to the image region or data statistics corresponding to the image region” via the AV1 Protocol, as described by at least the below technical specification(s). For example, the Accused Products identified in Appendix 1 practice the AV1 Specification, and, as shown below, the AV1 Specification specifies that at least one of the filters is adaptively enabled or disabled depending on data corresponding to the image region. Accordingly, the Accused Products, through operability with the AV1 Specification, practice Claim 7. <i>See also</i> Claim 5, which describes the underlying independent claim. <u>As described above with respect to Claim Element 5.2, the AV1 general tile group OBU syntax table includes a call to perform in-loop filtering, and as described above in Claim Element 5.3, the filtering is performed by invoking the ordered steps of (1) the loop filter process as in § 7.14 and (2) the CDEF process as in § 7.15. AV1’s loop filter is adaptively enabled or disabled depending on data characteristics or data statistics corresponding to the image region.</u> S1: § 7.14. Loop filter process S1: § 7.14.1 General Input to this process is the array CurrFrame of reconstructed samples. Output from this process is a modified array CurrFrame containing deblocked samples. The purpose of the loop filter is to eliminate (or at least reduce) visually objectionable artifacts associated with the semi- independence of the coding of super blocks and their constituent sub-blocks. The loop filter is applied on all vertical boundaries followed by all horizontal boundaries as follows:

CLAIM	PROTOCOL
	<pre data-bbox="457 272 1900 755"> for (plane = 0; plane < NumPlanes; plane++) { if (plane == 0 loop_filter_level[1 + plane]) { for (pass = 0; pass < 2; pass++) { rowStep = (plane == 0) ? 1 : (1 << subsampling_y) colStep = (plane == 0) ? 1 : (1 << subsampling_x) for (row = 0; row < MiRows; row += rowStep) for (col = 0; col < MiCols; col += colStep) loop_filter_edge(plane, pass, row, col) } } } </pre> <u>The edge loop filter process called by the loop filter is described in § 7.14.2, below:</u> S1: § 7.14.2. Edge loop filter process The inputs to this process are: • a variable plane specifying whether the process is filtering Y, U, or V samples, • a variable pass specifying the direction of the edges. pass equal to 0 means the process is filtering vertical block boundaries, and pass equal to 1 means the process is filtering horizontal block boundaries, • variables row and col specifying the location of the edge in units of 4x4 blocks in the luma plane. The outputs of this process are modified values in the array CurrFrame. (...) The variable applyFilter (equal to 1 if the samples are filtered) is derived as follows: • If isTxEdge is equal to 0, applyFilter is set equal to 0.

CLAIM	PROTOCOL
	- Otherwise, if isBlockEdge is equal to 1 or skip is equal to 0 or isIntra is equal to 1, applyFilter is set equal to 1. - Otherwise applyFilter is set equal to 0. The filter size process specified in section 7.14.3 is invoked with the inputs txSz, prevTxSz, pass, and plane, and the output assigned to the variable filterSize (containing the maximum filter size that can be used). The adaptive filter strength process specified in section 7.14.4 is invoked with the inputs row, col, plane, and pass, and the output assigned to the variables lvl, limit, blimit, and thresh. If lvl is equal to 0, the adaptive filter strength process specified in section 7.14.4 is invoked with the inputs prevRow, prevCol, plane, and pass, and the output assigned to the variables lvl, limit, blimit, and thresh. For the variable i taking values from 0 to MI_SIZE - 1, the following applies: - If applyFilter is equal to 1 and lvl is greater than zero, the sample filtering process specified in section 7.14.6 is invoked with the input variable x set equal to $xP + dy * i$, the input variable y set equal to $yP + dx * i$, and the variables plane, limit, blimit, thresh, dx, dy, and filterSize supplied as inputs. <u>The sample filter process called in the above edge loop filter process is described in § 7.14.6, below:</u> S1: § 7.14.6. Sample filtering process S1: § 7.14.6.1. General First the filter mask process specified in section 7.14.6.2 is invoked with the inputs x, y, plane, limit, blimit, thresh, dx, dy, and filterSize, and the output is assigned to the variables hevMask, filterMask, flatMask, and flatMask2. Then the appropriate filter process is invoked with the inputs x, y, plane, dx, dy as follows: - If filterMask is equal to 0, no filter is invoked. - Otherwise, if filterSize is equal to 4 or flatMask is equal to 0, the narrow filter process specified in section 7.14.6.3 is invoked with the additional input variable hevMask.

CLAIM	PROTOCOL
	• Otherwise, if filterSize is equal to 8 or flatMask2 is equal to 0, the wide filter process specified in section 7.14.6.4 is invoked with the additional input variable log2Size set to 3. • Otherwise, the wide filter process specified in section 7.14.6.4 is invoked with the additional input variable log2Size set to 4. <u>The filter mask process invoked in the above sample filtering process is described in § 7.14.6.2, below:</u> S1: § 7.14.6.2. Filter mask process The inputs to this process are: • variables x and y specifying the location within CurrFrame[plane], • a variable plane specifying whether the block is the Y, U or V plane, • variables limit, blimit, thresh that specify the strength of the filtering operation, • variables dx and dy specifying the direction perpendicular to the edge being filtered, • a variable filterSize of specifying the maximum size of filter allowed. The outputs from this process are the variables: • hevMask, • filterMask, • flatMask, (only used if filterSize >= 8), • flatMask2 (only used if filterSize >= 16). The values output for these masks depend on the differences between samples on either side of the specified boundary. These samples are specified as follows:

CLAIM	PROTOCOL
	<pre> q0 = CurrFrame[plane][y][x] q1 = CurrFrame[plane][y + dy][x + dx] q2 = CurrFrame[plane][y + dy * 2][x + dx * 2] q3 = CurrFrame[plane][y + dy * 3][x + dx * 3] q4 = CurrFrame[plane][y + dy * 4][x + dx * 4] q5 = CurrFrame[plane][y + dy * 5][x + dx * 5] q6 = CurrFrame[plane][y + dy * 6][x + dx * 6] p0 = CurrFrame[plane][y - dy][x - dx] p1 = CurrFrame[plane][y - dy * 2][x - dx * 2] p2 = CurrFrame[plane][y - dy * 3][x - dx * 3] p3 = CurrFrame[plane][y - dy * 4][x - dx * 4] p4 = CurrFrame[plane][y - dy * 5][x - dx * 5] p5 = CurrFrame[plane][y - dy * 6][x - dx * 6] p6 = CurrFrame[plane][y - dy * 7][x - dx * 7] </pre> Note: Samples q4, q5, q6, p4, p5, and p6 are only used if filterSize is equal to 16. (...) The value of filterMask indicates whether adjacent samples close to the edge (within four samples either side of the specified boundary) vary by less than the limits given by limit and blimit. It is used to determine if any filtering should occur and is calculated as follows:

CLAIM	PROTOCOL
	<pre> limitBd = limit << (BitDepth - 8) blimitBd = blimit << (BitDepth - 8) mask = 0 mask = (Abs(p1 - p0) > limitBd) mask = (Abs(q1 - q0) > limitBd) mask = (Abs(p0 - q0) * 2 + Abs(p1 - q1) / 2 > blimitBd) if (filterLen >= 6) { mask = (Abs(p2 - p1) > limitBd) mask = (Abs(q2 - q1) > limitBd) } if (filterLen >= 8) { mask = (Abs(p3 - p2) > limitBd) mask = (Abs(q3 - q2) > limitBd) } filterMask = (mask == 0) </pre>
8. The apparatus of claim 5, wherein a version of the image region that is displayed is different from at least one of an output signal corresponding to the image region after the at least two in-loop filters and an initial input signal corresponding to the image region before	The Accused Products either literally and/or under the doctrine of equivalents practice the claim “a version of the image region that is displayed is different from at least one of an output signal corresponding to the image region after the at least two in-loop filters and an initial input signal corresponding to the image region before the at least two in-loop filters” via the AV1 Protocol, as described by at least the below technical specification(s). For example, the Accused Products identified in Appendix 1 practice the AV1 Specification, and, as shown below, the AV1 Specification has two operating modes: the first is general decoding where the input is a sequence of OBUs and the output is decoded frames, which is specified in the general decoding process of the protocol. The general decoding process § 7.2 then specifies that when <code>film_grain_params_present</code> is equal to 0, the decoders produce output frames according to the general decoding process detailed below, while the decoder output when <code>film_grains_params_present</code> is equal to 1 is modified using a film grain synthesis process. Accordingly, the Accused Products, through operability with the AV1 Specification, practice Claim 8. <i>See also</i> Claim 5, which describes the underlying independent claim.

CLAIM	PROTOCOL
the at least two in-loop filters.	<u>As described above with respect to Claim Element 5.1, the AV1 bitstream two operating modes: the first is general decoding where the input is a sequence of OBUs and the output is decoded frames, which is specified in the general decoding process of the protocol, as described in § 7.2, below:</u> S1: § 7.2. General decoding process When film_grain_params_present is equal to 0, decoders shall produce output frames that are identical in all respects and have the same output order as those produced by the decoding process specified herein. When film_grain_params_present is equal to 1, a decoder shall implement a film grain synthesis process that modifies the output arrays OutY, OutU, OutV. The reference film grain synthesis process is described in section 7.18.3. <u>As described above with respect to Claim Element 1.2, the decode_frame_wrapup process performs in-loop filtering using at least two filters in succession, relevant portions of the decode frame wrapup process are described in § 7.4, below:</u> S1: § 7.4. Decode frame wrapup process This process is triggered by a call to decode_frame_wrapup from within the syntax tables. At this stage, all the tile level decode has been done, and this process performs any frame level decode that is required. If show_existing_frame is equal to 0, the process first performs any post processing filtering by the following ordered steps: 1. If loop_filter_level[0] is not equal to 0 or loop_filter_level[1] is not equal to 0, the loop filter process specified in section 7.14 is invoked (this process modifies the contents of CurrFrame). 2. The CDEF process specified in section 7.15 is invoked (this process takes CurrFrame and produces CdefFrame). (...) The following ordered steps now apply: 1. The reference frame update process as specified in section 7.20 is invoked (this process saves the current frame state into the reference frames).

CLAIM	PROTOCOL
	2. If show_frame is equal to 1 or show_existing_frame is equal to 1, the output process as specified in section 7.18 is invoked (this will output the current frame or a saved frame). <u>As described above with respect to Claim Element 5.2, the AV1 loop filter process is applied, and relevant portions of the loop filter process are described in § 7.14, below:</u> S1: § 7.14. Loop filter process S1: § 7.14.1. General Input to this process is the array CurrFrame of reconstructed samples. Output from this process is a modified array CurrFrame containing deblocked samples. <u>As described above with respect to Claim Element 1.3, the AV1 loop filter process is applied, and relevant portions of the CDEF process are described in § 7.15, below:</u> S1: § 7.15. CDEF process Input to this process is the array CurrFrame of reconstructed samples. Output from this process is the array CdefFrame containing deringed samples. <u>The output process called in the decode frame wrapup process when show_frame is equal to 1 or show_existing_frame is equal to 1 is described in § 7.18, below:</u> S1: § 7.18. Output process S1: § 7.18.1. General This process is invoked to prepare output frames. (...) The intermediate output preparation process specified in section 7.18.2 is invoked to prepare arrays OutY, OutU, and OutV, and the outputs are assigned to w, h, subX, and subY.

CLAIM	PROTOCOL
	If film_grain_params_present is equal to 1 and apply_grain is equal to 1, then the film grain synthesis process specified in section 7.18.3 is invoked with inputs of w, h, subX, and subY. (This process modifies the output arrays OutY, OutU, OutV). Finally, the frame to be output is defined to be the arrays OutY, OutU, OutV where the bit depth for each sample is BitDepth. This frame to be output is the overall output of the decoding process and further processing (such as color conversion) is outside the scope of this specification. For example, a real implementation might use these arrays to display the frame to the user, or a test system might save the arrays so the output can be verified.