OpenCores.Org

EV_JPEG_ENC IP Core Specification

Author: Michal Krepa

Rev. 2.3
February 25, 2021
Krakow, PL

Version	Date	Author	Comment
1.0	15.02.2009	Michal Krepa	Initial Creation.
1.1	13.03.2009		Huffman description updated.
1.2	17.03.2009		Huffman design restored to use two stage VLI/VLC handler to achieve better timing. Simulation HOWTO chapter added.
1.3	19.03.2009		BUF_FIFO fifo almost full signal changed.
1.4	20.03.2009		Added C_MEMORY_OPTIMIZED configuration constant to reduce memory footprint at expense of performance. Look at BUF_FIFO description for details.
1.5	24.03.2009		BUF_FIFO now uses single RAM which is used for all SubFIFOs. Added description of divider used in Quantizer block.
1.6	26.03.2009		Added chrominance quantization table support.
1.7	28.03.2009		Added separate pipeline stage for Quantizer to balance chain load across design.
1.8	29.03.2009		Added chrominance table support for Huffman encoding. Fixed bug in RLE for ZRL (Zero Run Length) handling. Fixed bug in number of encoded bytes output.
1.9	02.05.2009		Fixed bugs in BUF_FIFO almost full flag generation.
2.0	29.10.2009		Added output interface backpressure capability.
2.1	11.11.2009		Added 16 bit RGB 565 format.
2.2	27.11.2009		Changed BUF_FIFO design.
2.3	29.11.2009		Design modified to use 4:2:2 chroma subsampling

1.1 EV_JPEG_ENC

1.1.1 General description

EV_JPEG_ENC core is intended to encode raw bitmap images into JPEG compliant coded bit stream. JPEG baseline encoding method is used.

1.1.2 Architecture

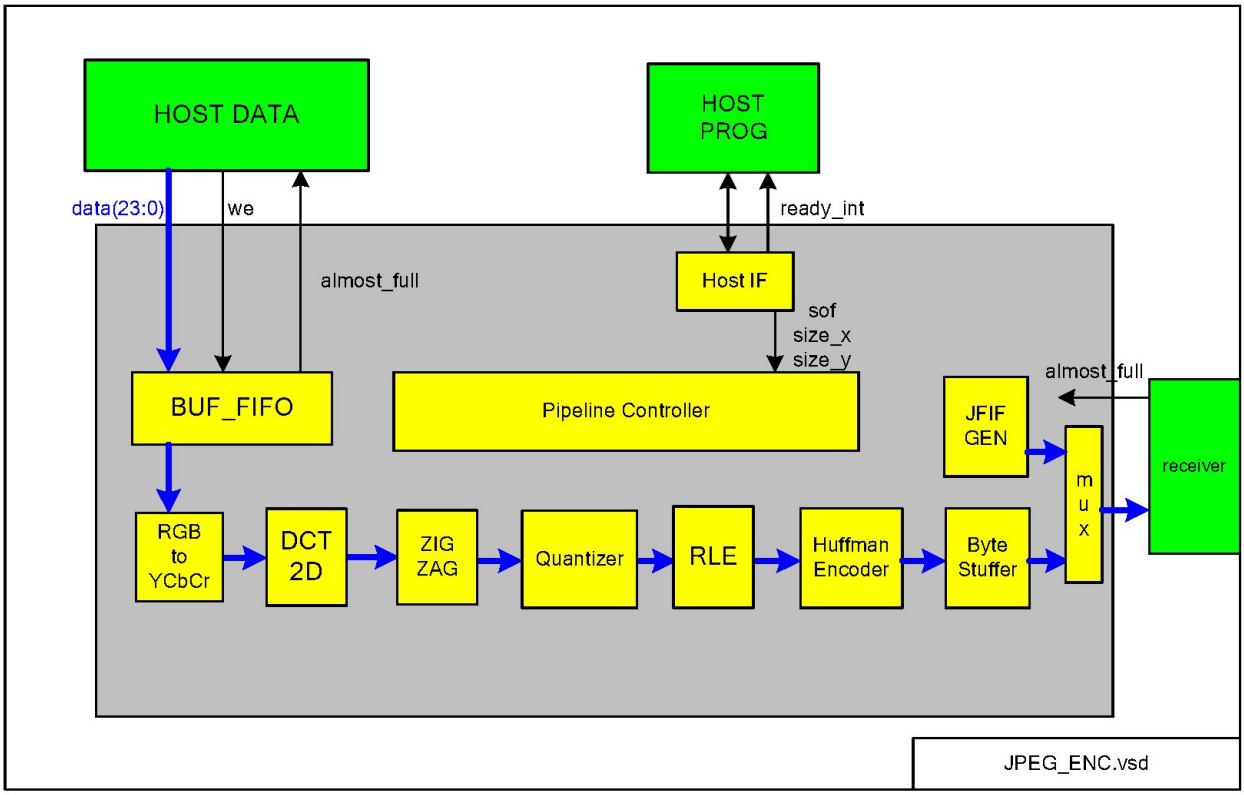

Figure 1

General system architecture consist of encoding chain started by Host Programming interface. Host Data interface shall continuously write BUF_FIFO until FIFO almost full signal is received. Then, it should stop and wait for signal FIFO almost full to deassert. When this is the case it should continue writing and so on.

Encoding is governed by Controller and is a pipelined process where each pipeline stage process 16x8 block of samples at a time (16x8 block is so called “data unit”).

Finally, encoded bit stream is byte stuffed and then stored to output (RAM or FIFO).

Following steps are performed: Line Buffering, Chroma subsampling, RGB to YCbCr conversion, Discrete Cosine Transform 2-dimensional, followed by Zig Zag scan, Quantizer, Run Length Encoding and Huffman coding followed by byte stuffing and JFIF header is also generated at the beginning of encoding and with EOI marker closing image.

1.1.3 Features

- JPEG baseline encoding JPEG ITU-T T.81 | ISO/IEC 10918-1
- Standard JFIF header v 1.01 automatic generation
- Color images only (3 components, RGB input)

- Two programmable Quantization tables, one for luminance and one for chrominance
- Hardcoded Huffman tables
- under 2.7 clock cycles per one input 24 bit pixel @ 50% quality setting
- OPB programming and data Host interface
- 4:2:2 Chroma subsampling
- Source code target independent, synthesizable RTL VHDL code
- Design easily exceeds 100 MHz on decent FPGA like Stratix II, Virtex IV etc.
- Stallable output interface
- supports 24 bit input RGB or 16 bit input RGB 565

1.1.4 Example throughput

Measured from JPEG encoding start till encoding done:

- Input image 640x480 24 bit RGB color. New sample loaded every cycle until FIFO full.
- Quantization tables at 50% quality setting
- **7.3 ms processing time @ 100 MHz clock [2.3 clock cycles per input sample]**
- **1000/7.3=136 frames per second @ 100 MHz**
- Input file size = 921 kB. Output file size = 44 kB (depends on image)
- Compression stats (from JPEGSnoop software):
  - Compression Ratio: 21.31:1
  - Bits per pixel: 1.13:1

1.1.5 Control State Machine (CTRL_SM)

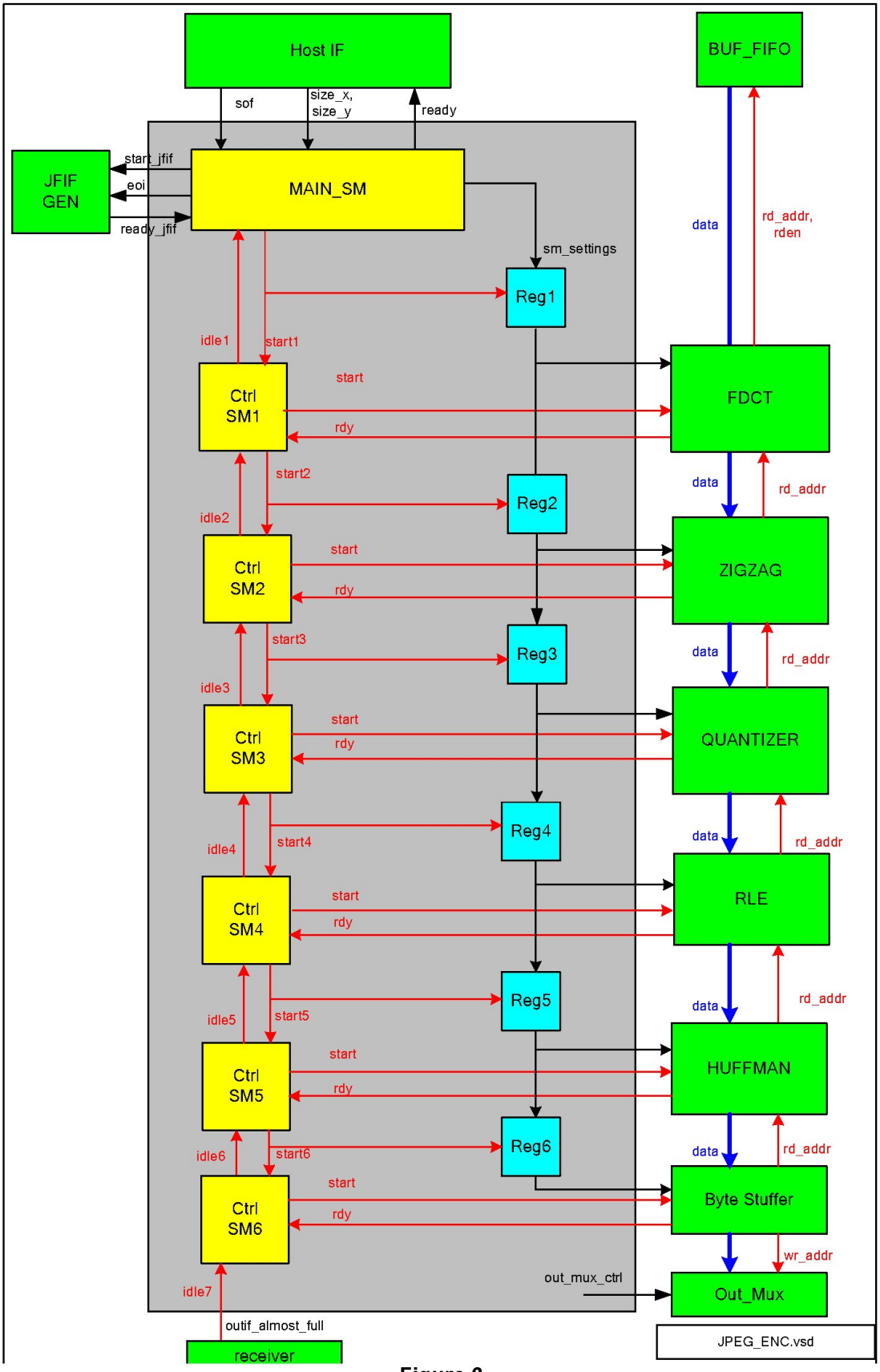

Figure 2

Description:

- EV_JPEG_ENC is main control state machine for JPEG encoding process. It gets uncoded raw image samples and transforms it into JPEG encoded bit stream.
- First, Host asserts “sof” (start of frame) single pulse along with size_x (image width) and size_y (image height) signals valid (programming registers are used for that purpose).
- After this is done, Host starts loading raw image to Host Data interface
- Then EV_JPEG_ENC process image in 8x8 pixels blocks following horizontal order, from left to right and vertically from top to bottom of the image. When color image is encoded, 4 components (Y1, Y2, Cb, Cr) are interleaved 8x8 block wise.
- 4:2:2 subsampling is used which implies extracting from each 16x8 block four components, two luminance and two chrominance. Chrominance uses 2:1 horizontal subsampling
- Limitation: Only image sizes (height, width) multiple of (8,16) samples are supported
- Finally, coded bit stream is stored to output RAM.
- Ready signal to Host is asserted when all 8x8 blocks are processed and all pipeline state machines are idle. Also, ready_int interrupt to Host is generated.
- Module receiving encoded JPEG stream “receiver” can stall JPEG core by asserting outif_almost_full signal. This can be used in cases where receiver is not able to keep up with JPEG encoder data rate. In such case, receiver asserts almost full flag when it’s internal buffer has only 64+X empty bytes where X is a kind of safety margin. Margin can be set to 8-16.

1.1.6 MAIN_SM

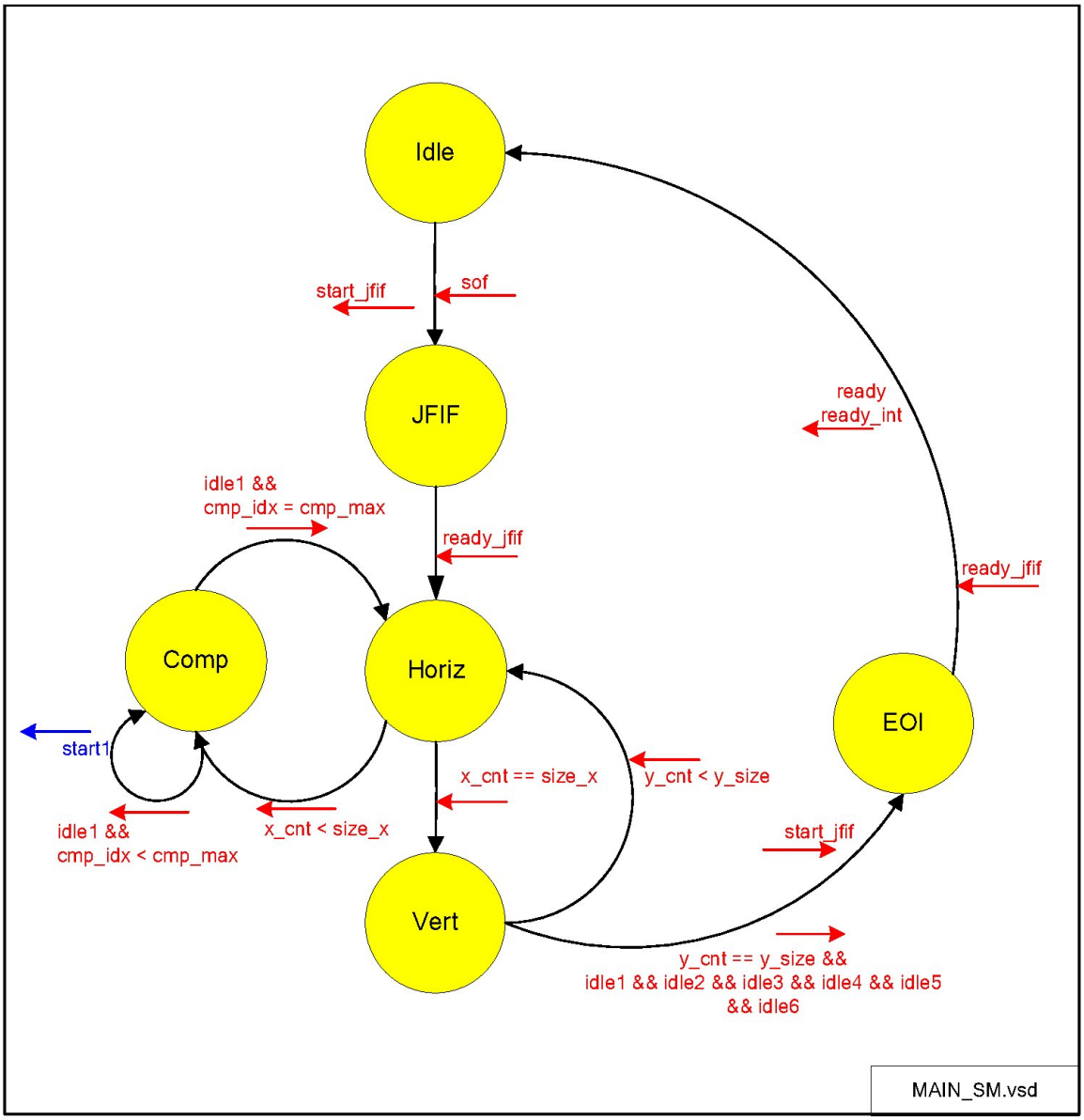

Figure 3

MAIN_SM state machine is responsible for control of whole encoding process.

1.1.6.1 Some variables/signals are defined

Name	description	range
x_cnt	count samples in horizontal direction. Incremented in steps of 16 samples. (16 bits)	0..image_width-16 **warning**: image width must be multiple of 16 samples
y_cnt	count lines in vertical direction. Incremented in steps of 8 samples. (16 bits)	0..image_height-8 **warning**: image height must be multiple of 8 samples
cmp_idx	Component index. Indicates currently processed color subsampling component (Y1, Y2, Cb, Cr).	0..3
sof	start of frame. Asserted by Host after raw	0/1

	(uncoded) image is loaded.	
size_x	image width (16 bits)	8...65536-16 (limited by RAM resources)
size_y	image height (16 bits)	8...65536-8
start1	starts first stage of pipeline (FDCT)	0/1
ready	ready pulse signal to Host – JPEG encoding complete	0/1
cmp_max	Maximum number of components.	4
eoi	generation of EOI marker to end coded image data. Sampled when jfif_start is high.	0/1

1.1.6.2 States description

Name	description
Idle	JPEG encoder is inactive waiting to be started by Host
Horiz	encoding process blocks of 16x8 samples until horizontal counter reaches width of image.
Vert	Reset horizontal counter and advance vertical counter by 8. Check if end of image reached.
Comp	Component processing. Process one 16x8 block from each component in interleaved fashion including subsampling
JFIF	Generation of JFIF header.
EOI	Generation of EOI marker (End Of Image)

1.1.6.3 SM_SETTINGS

SM_SETTINGS is record passed through pipeline from main state machine. When start1 asserts sm_settings from MAIN_SM are latched to Reg1.

SM_SETTINGS	description
x_cnt	current sample coordinate starting point in horizontal direction
y_cnt	current sample coordinate starting point in vertical direction
cmp_idx	Component index.

Order of processing:

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
10	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**
20	**21**	**22**	**23**	**24**	**...**	**...**	**...**	**...**	**...**

Each cell in table above represents block 16x8 pixels (“data unit”). Blocks are processed in direction of increasing numbers.

1.1.7 MAIN_SM details

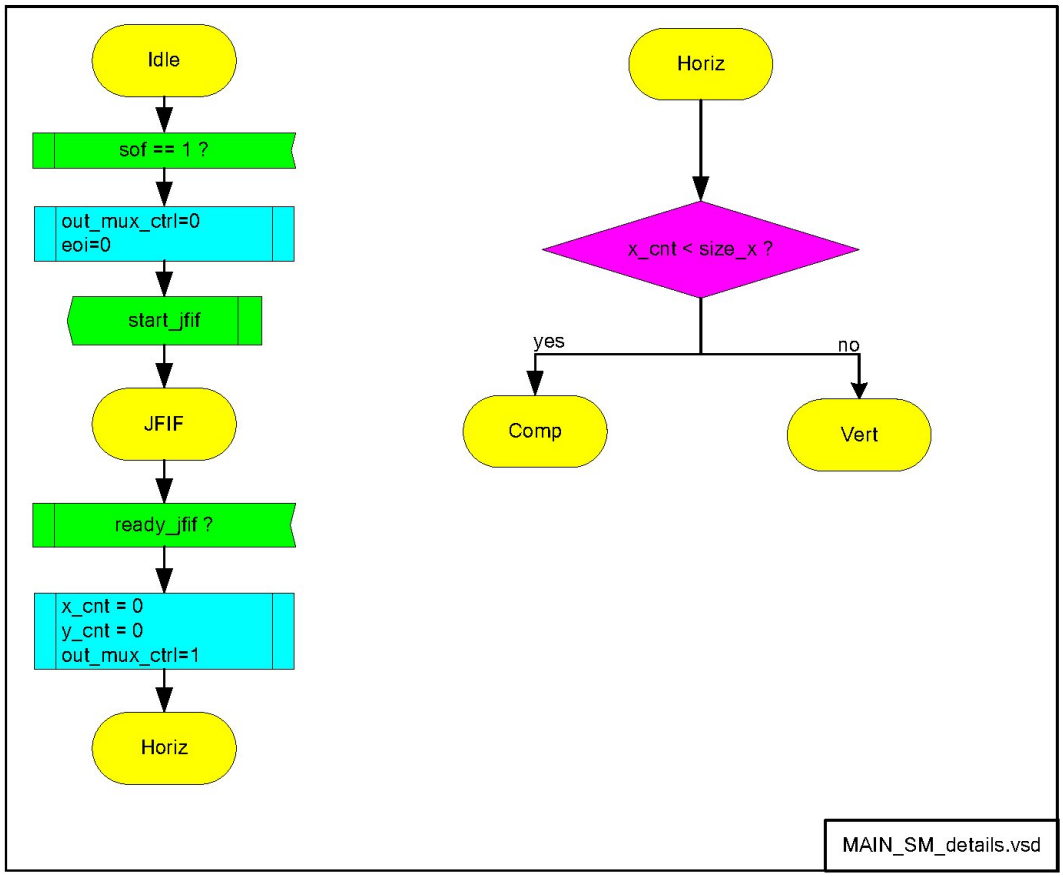
Idle
sof == 1 ?
out_mux_ctrl=0
eoi=0
start_jfif
JFIF
ready_jfif ?
x_cnt = 0
y_cnt = 0
out_mux_ctrl=1
Horiz
Horiz
x_cnt < size_x ?
yes
no
Comp
Vert
MAIN_SM_details.vsd

Figure 4

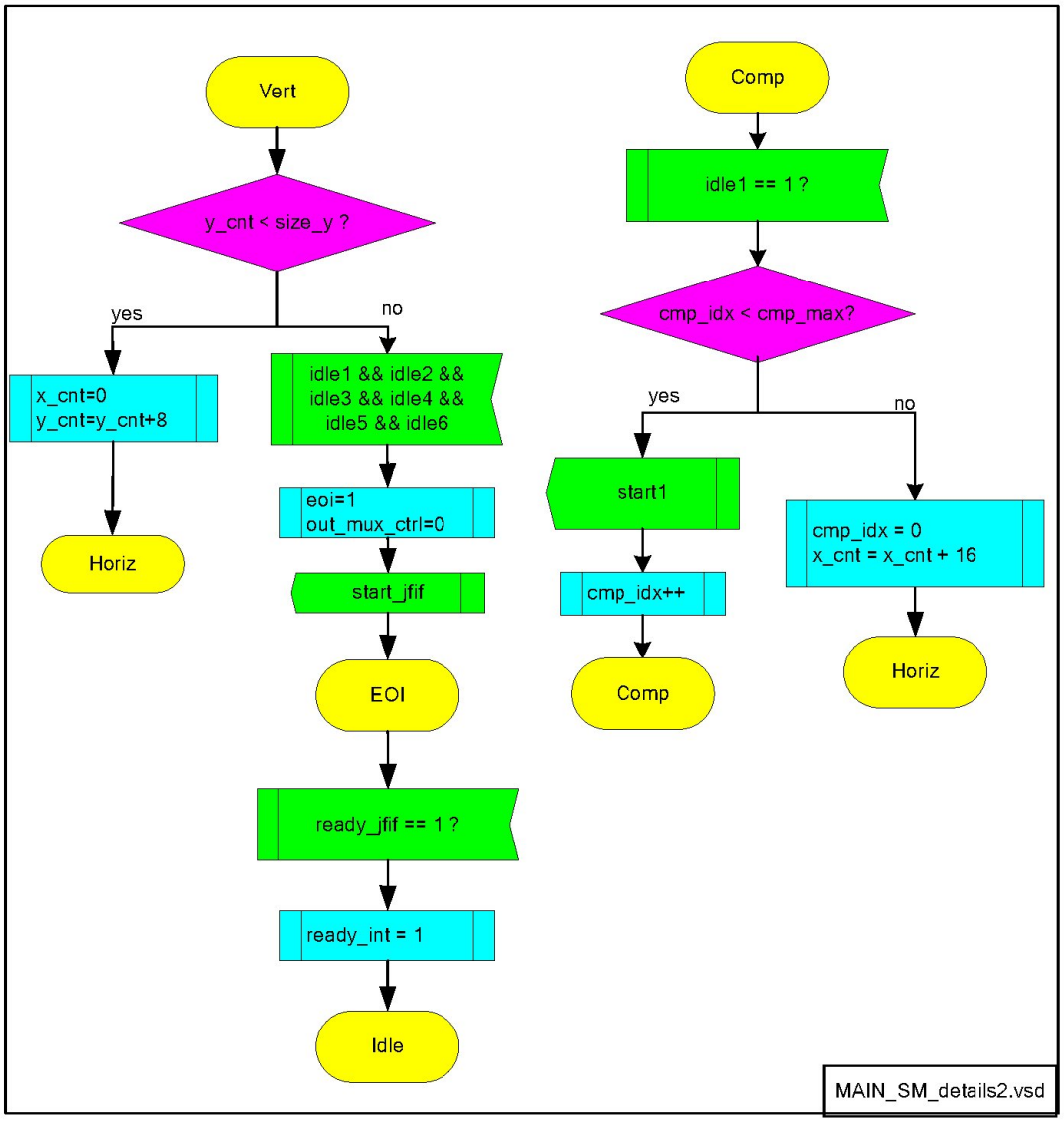
Vert
y_cnt < size_y ?
yes
no
x_cnt=0
y_cnt=y_cnt+8
Horiz
idle1 && idle2 &&
idle3 && idle4 &&
idle5 && idle6
eoi=1
out_mux_ctrl=0
start_jfif
EOI
ready_jfif == 1 ?
ready_int = 1
Idle
Comp
idle1 == 1 ?
cmp_idx < cmp_max?
yes
no
start1
cmp_idx++
Comp
cmp_idx = 0
x_cnt = x_cnt + 16
Horiz
MAIN_SM_details2.vsd

Figure 5

1.2 BUF_FIFO

Host Data interface writes input image line by line to BUF_FIFO. This FIFO is intended to minimize latency between raw image loading and encoding start. It performs raster to block conversion (line wise input to 8x8 block conversion).

BUF_FIFO is actually a RAM.

Note: Maximum supported image width must be configured by constant in JPEG_PKG.VHD to fix RAM size before synthesis.

Default **C_MAX_LINE_WIDTH = 640**

Warning: This parameter heavily affects size of used memory (block ram etc).

BUF_FIFO must be able to store at least 8 lines of input image, so that JPEG encoding can be started for 8x8 blocks.

RAM size is configurable via VHDL constant C_EXTRA_LINES in JPEG_PKG.VHD from 8 to 16 lines with 1 line step. 16 lines buffer gives highest speed while 8 lines gives lowest area.

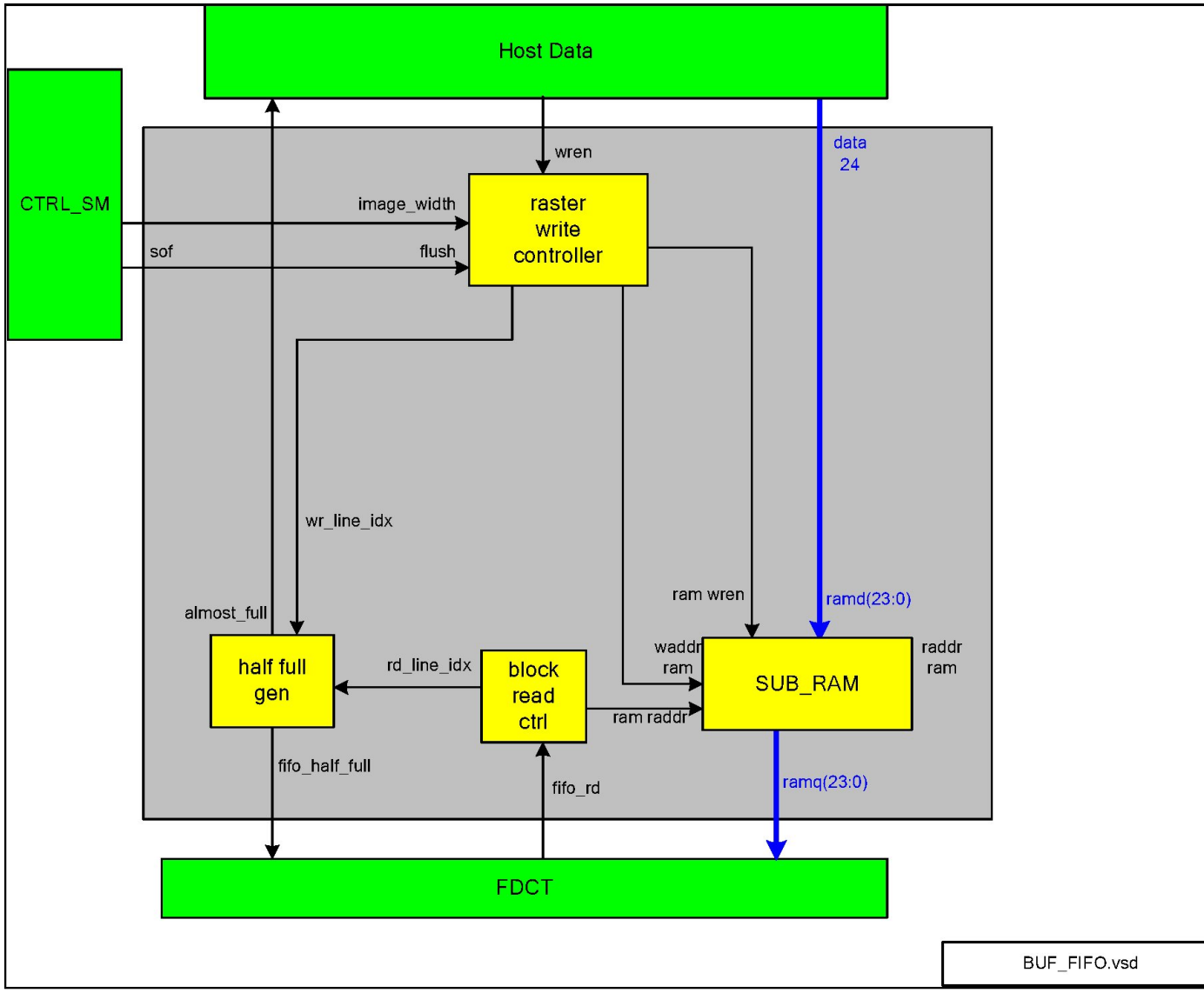

Figure 6

1.2.1.1 RAM write

- Host writes image data line by line.
- RAM is written from 0 to n-1 (left to right) with 8 pixels in each line written per FIFO
- pixel_cnt(15:0) is incremented every write enable from Host and is reset when end of line is reached (equal to image_width-1).

1.2.1.2 RAM read

- RAM is being read 8x8 block wise from left to right, top to bottom.

1.2.1.3 General

1.2.1.4 SUB_RAM

SUB_RAM is memory used as line buffer.

1.2.2 Output Mux

Output Mux is used to switch data path to JFIF Generator for the time of JFIF header writing. After JFIF header is written to output, mux is switched to “normal” data processing path.
Switch is realized by means of signal out_mux_ctrl toggle.

1.3 Host IF

Host IF is an interface block between host (CPU) and JPEG encoder IP core.
Xilinx OPB v2.1 has been selected as Host communication protocol. JPEG encoder core will act as OPB slave device.

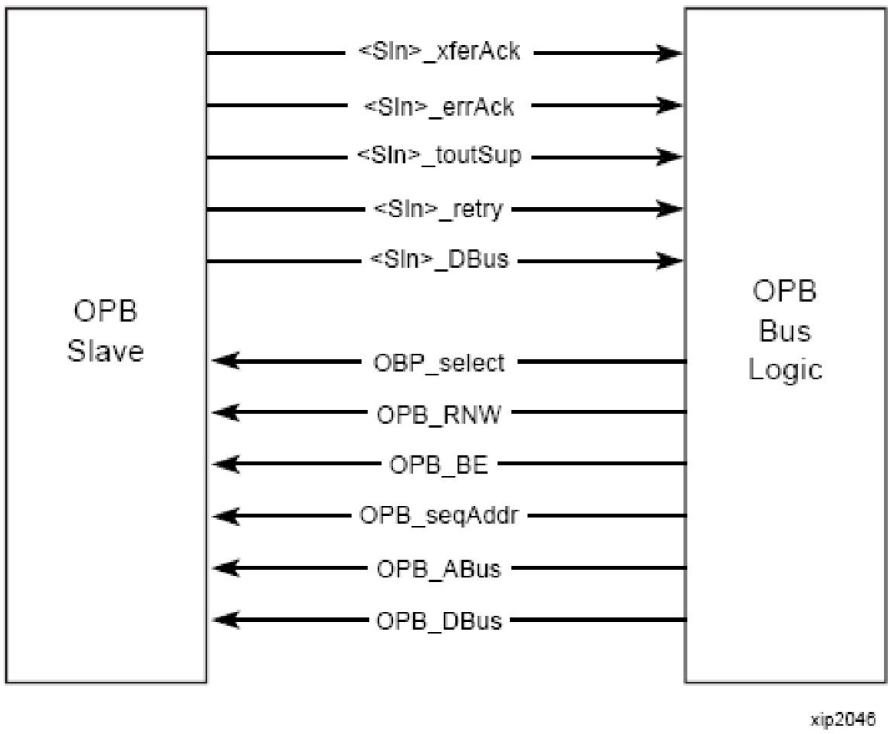

Figure 7

More information on interface can be found in OPB v2.1 specification.

On figure below OPB read transaction is shown

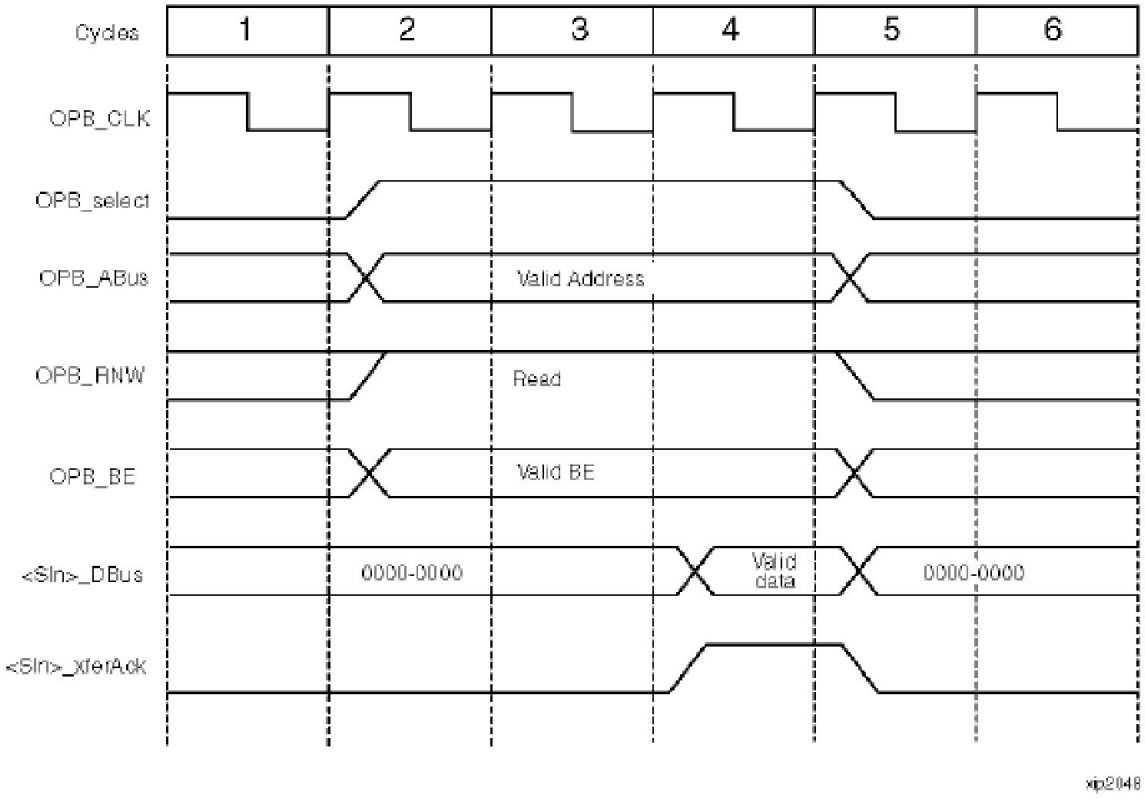

Figure 8

On figure below OPB write transaction is shown.

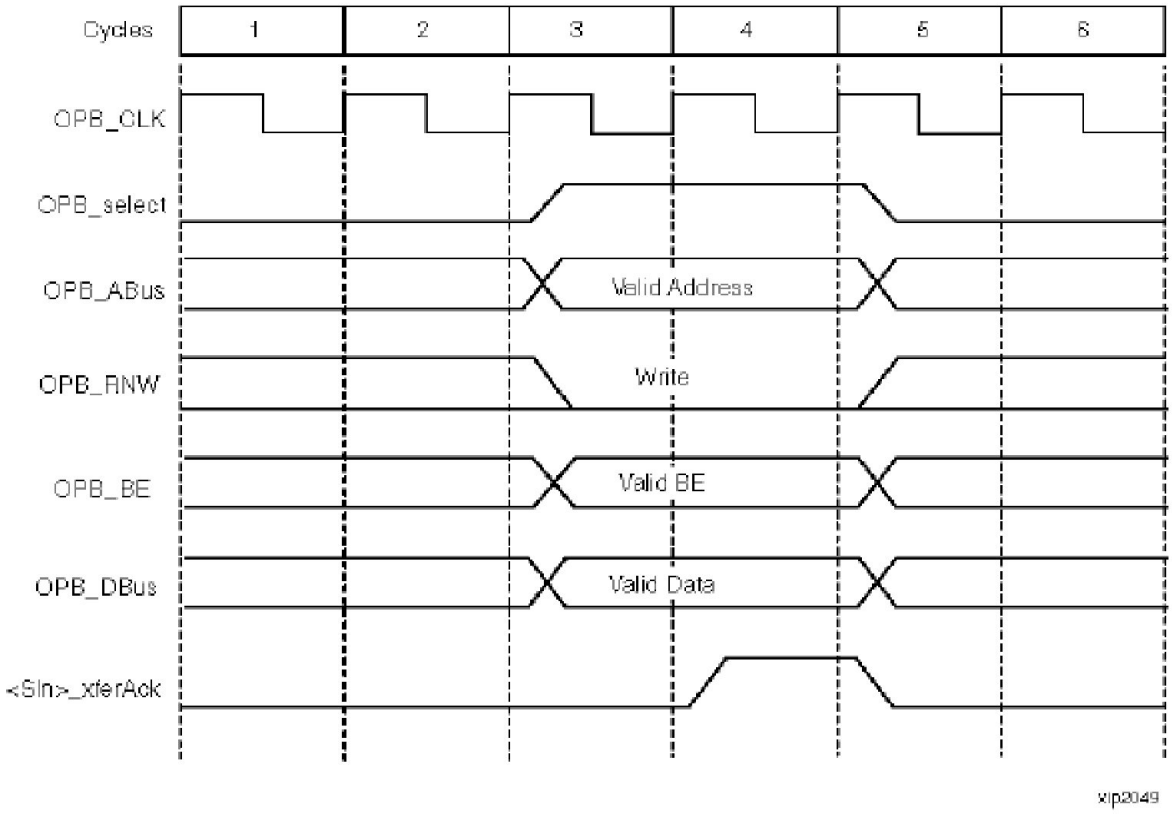

Figure 9

Note: Above images are taken from “Designing Custom OPB Slave Peripherals for MicroBlaze” Xilinx tutorial.

1.4 FDCT

FDCT is intended to perform functions: RGB to YCbCr conversion, chroma subsampling, input level shift and DCT (discrete cosine transform).

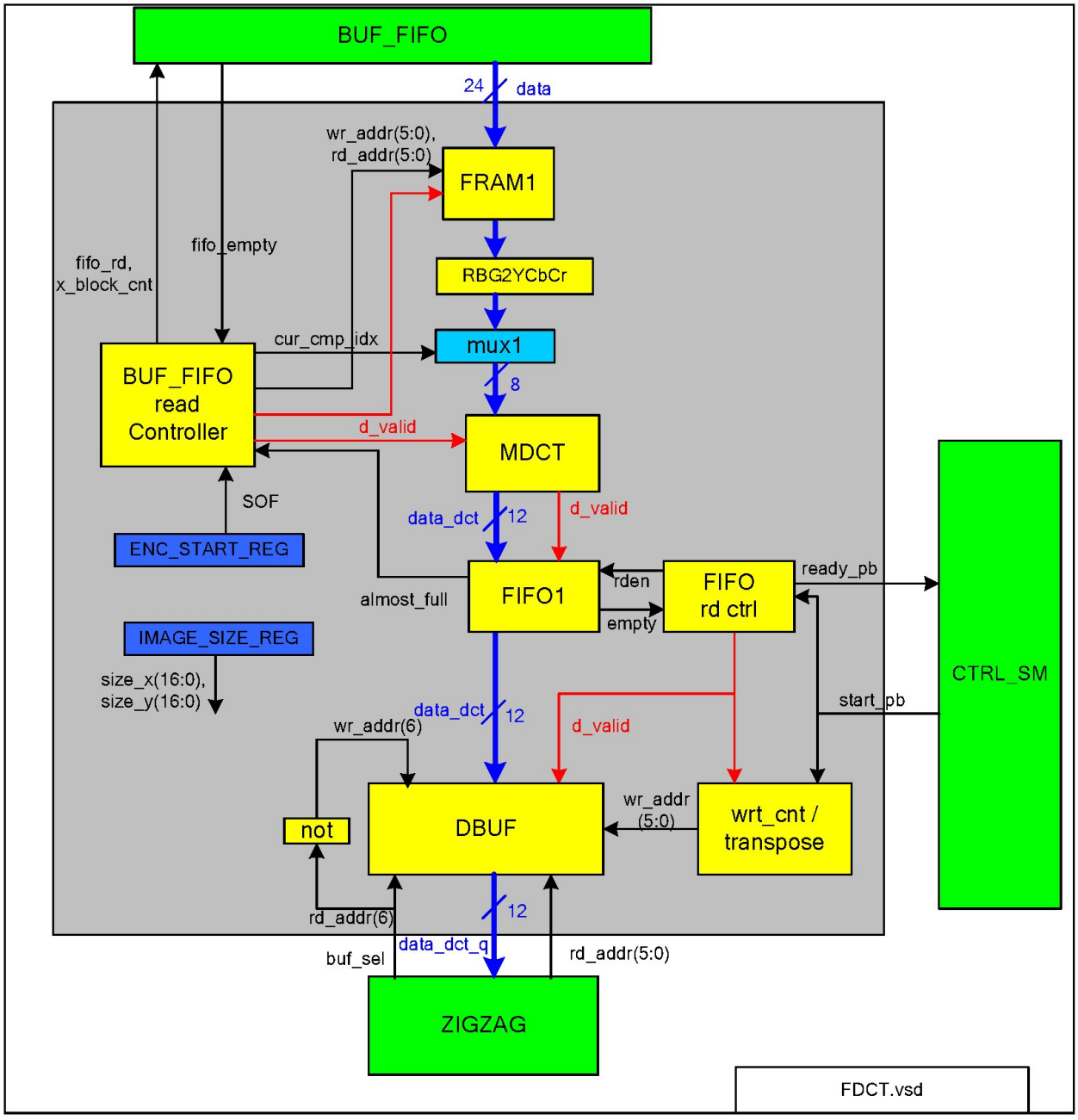

Figure 10

When sof asserts

- size_x and size_y are latched internally
- RAM Read Ctrl is started

1.4.1 BUF_FIFO Read Controller

This submodule performs reading of input samples fom BUF_FIFO. To maximize performance this block operates “almost” independently from EV_JPEG_ENC pipeline control processing chain.

- When SOF asserts, Read Controller starts reading data from BUF_FIFO given it is not empty.
- Read Ctrl has its own horizontal, vertical and component counters to point into currently processed 16x8 block and know end of image. These counters operate similarly to ones in Pipeline Controller FSM.
- Independent counters are necessary as Read Ctrl can operate at higher rate than rest of processing chain.
- Data read from BUF_FIFO is given to MDCT core
- Read Ctrl stalls reading when FIFO1 has less space than 64 words (aka FIFO1 almost full signal). This is necessary to prevent FIFO1 overflow, because MDCT output results in bursts of 64 words.
- Rad Ctrl stalls reading when BUF_FIFO is empty
- Read_ctrl is started once by SOF and then it does self-trigger internal start after reading of one 16x8 block.
- When all 16x8 blocks are read out internal self-start mechanism is stopped and Read Ctrl goes to idle.
- Reading of one 16x8 block proceeds like follow: read one 16x8 block of pixels from BUF_FIFO and store it to FRAM1. Read FRAM1 4 times one per each component to perform RGB to YCbCr conversion, subsampling and DCT
- cur_cmp_idx is counter modulo 4. It is incremented every time RAM RD CTRL is started for new 16x8 block.

Reading details:

- 64 image samples are read from FRAM1 using rd_addr as read address. These contain R,G,B 8 bit values per one sample. This is done 4 times, one per each component. At each pass, different data (Y1,Y2 Cb, Cr) is taken to DCT processing.
- First, 64 Y1 values undergo DCT from left 8x8 block (increment cur_cmp_idx), then 64 Y2 from right 8x8 block(increment cur_cmp_idx), 64 Cb subsampled and finally 64 Cr subsampled. Thus 4 x 64 x 8 bit data from Image RAM is level shifted and DCT encoded
- chroma subsampling in first version will be realized as taking every 2nd horizontal pixel from 16x8 block which will effect in 8x8 subsampled block (no averaging)

1.4.2 FRAM1

FRAM1 is dual port memory with size 128x24 bits. Used to store one 16x8 block read from BUF FIFO. Since one 16x8 block read from FIFO contains data already for all 4 components and 4 components are processed in 4 consecutive read passes, it means reading 4 times the same 24 bit data is necessary, just use different byte from it each time. As from FIFO we cannot read 4 times same data, intermediate RAM=FRAM1 is used.

1.4.3 RGB to YCbCr conversion

Following equation is implemented by means of multipliers and adders to perform conversion:

```
Y  = (0.299*R)+(0.587*G)+(0.114*B)
Cb = (-0.1687*R)-(0.3313*G)+(0.5*B)+128
Cr = (0.5*R)-(0.4187*G)-(0.0813*B)+128
```

Constants used in this equation have format 14 bits of precision plus 1 sign bit on MSB.

Saturation to 255 is not implemented. Not necessary it seems???

Warning: EV_JPEG_ENC design assumes that hardware multipliers are available on target FPGA!

1.4.4 Write Counter / DCT matrix transpose

- wrt_cnt is write counter used to count number of samples (0..63
- Every time d_valid output from MDCT asserts, wrt_cnt is post incremented by 1.
- Wrt_cnt is reset by start_pb to zero.
- When all words are written (wrt_cnt == 63) ready_pb asserts to signal block processing complete to EV_JPEG_ENC chain control.

- It is necessary to transpose MDCT output as it is column wise output while row wise is needed.
- For that purpose two 3 bit counters are created. yw_cnt counts vertically and xw_cnt counts horizontally. Normally, yw_cnt counts from 0..7, when 7 is reached xw_cnt is incremented and yw_cnt reset. xw_cnt is wrap around type of counter. Write address to DBUF: wraddr(5:3) = yw_cnt, wraddr(2:0) = xw_cnt

1.4.5 Mux1

Mux1 is used to select currently processed image component. It routes proper byte from 24 bits input data. Data(7:0) is used for cur_cmp_idx=0,1, data(15:8) for cur_cmp_idx=2, data(23:16) for cur_cmp_idx=3.

1.4.6 MDCT

2D Discrete cosine transform core combined with level shift. Works on block of 64 samples. Take 8 bit input and produces 12 bit output.

First, uncoded image is level shifted from unsigned integers with range [0, 2^P - 1] to signed integers with range [-2^(P-1), 2^(P-1)-1]. x^p means here x to power of p.

Then 2D DCT is performed using following equation:

$$X(u,\ v) = \frac{2}{N} \cdot C(u)\ \cdot C(v)\ \cdot \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} x(i,\ j)\ \cdot \cos\frac{(2i\ +1)u\ \pi}{2N} \cdot \cos\frac{(2j\ +1)v\ \pi}{2N}$$

- where C(u),C(v) $= 2^{-1/2}$ for u=0,v=0
  $= 1$ otherwise
- x(i,j) – input sample at position (i,j) in 8x8 block
- X(u,v) – output sample at position (u,v) in 8x8 block
- N=64

MDCT takes data row-wise but outputs column-wise. To get row-wise order it is necessary to transpose output DCT matrix.

1.4.7 FIFO1 and FIFO RD CTRL

FIFO1 has space for 5 8x8 block DCT results. Its size = 5x64x12 bit.
FIFO read controller has internal counter fifo_cnt(6:0) to count number of words read per one start_pb. When start_pb asserts:

- fifo_cnt is reset and starts FIFO reading.
- if fifo_cnt is less than 64 and FIFO is not empty -> read FIFO and increment fifo_cnt
- when fifo_cnt is = 64-1 FIFO reading is disabled until next start_pb

1.4.8 DBUF

Dual port RAM 2x64x12 bit. Double Buffer necessary for pipeline operation. Buf_sel signal is used as MSB in read address of DBUF. ~buf_sel (inverted) is used as MSB of write address to DBUF. Thus, while FDCT is writing
next block of data, ZIGZAG is reading previous block.

1.5 ZIGZAG

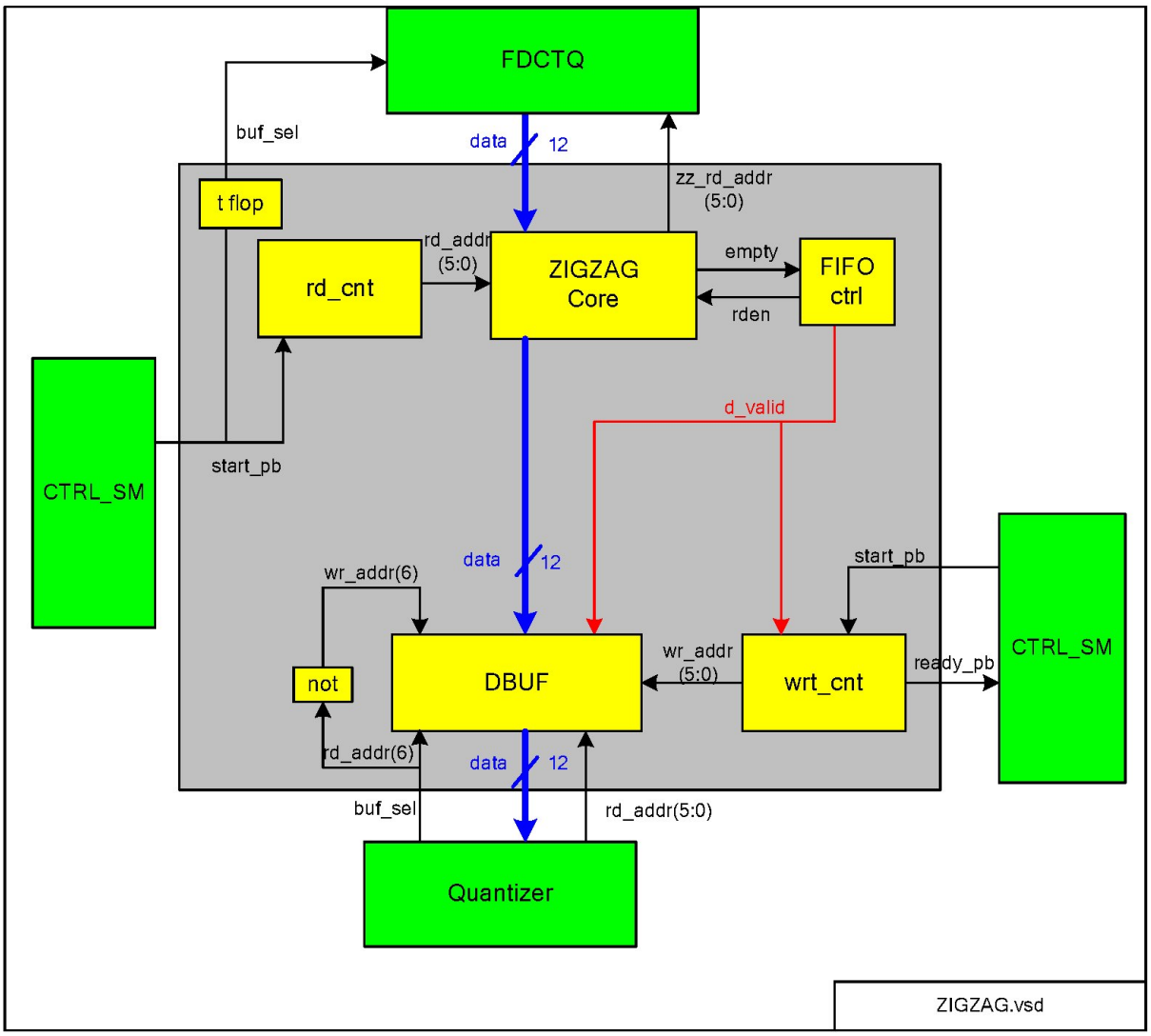

Figure 11

Zig-Zag block is responsible to perform so called zig-zag scan. It is simply reorder of samples positions in one 8x8 block according to following tables.

input order (natural):

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	...	...	
				...	...	62	63

Zig-zag output order:

0	1	5	6	14	15	27	28
2	4	7	13	16	26	29	42
3	8	12	17	25	30	41	43
9	11	18	24	31	40	44	53
10	19	23	32	39	45	52	54
20	22	33	38	46	51	55	60
21	34	37	47	50	56	59	61
35	36	48	49	57	58	62	63

It means for example that first sample (position 0) is mapped to the same output position 0. Sample 2 is mapped to output position 5, etc.

When start_pb asserts:

- buf_sel toggles
- wrt_cnt is reset
- rd_cnt is reset to 0 and starts counting 0..63. -> 64 values are read from FDCT.
- ZIGZAG scanning is performed
- reordered data is written to DBUF using wr_addr generated from write counter (wrt_cnt)
- after 64 values are written to DBUF ready_pb is asserted
- Wrt_cnt can be implemented as simply delayed version of rd_cnt as ZIGZAG scan processing latency is constant.

1.5.1 ZIGZAG Core

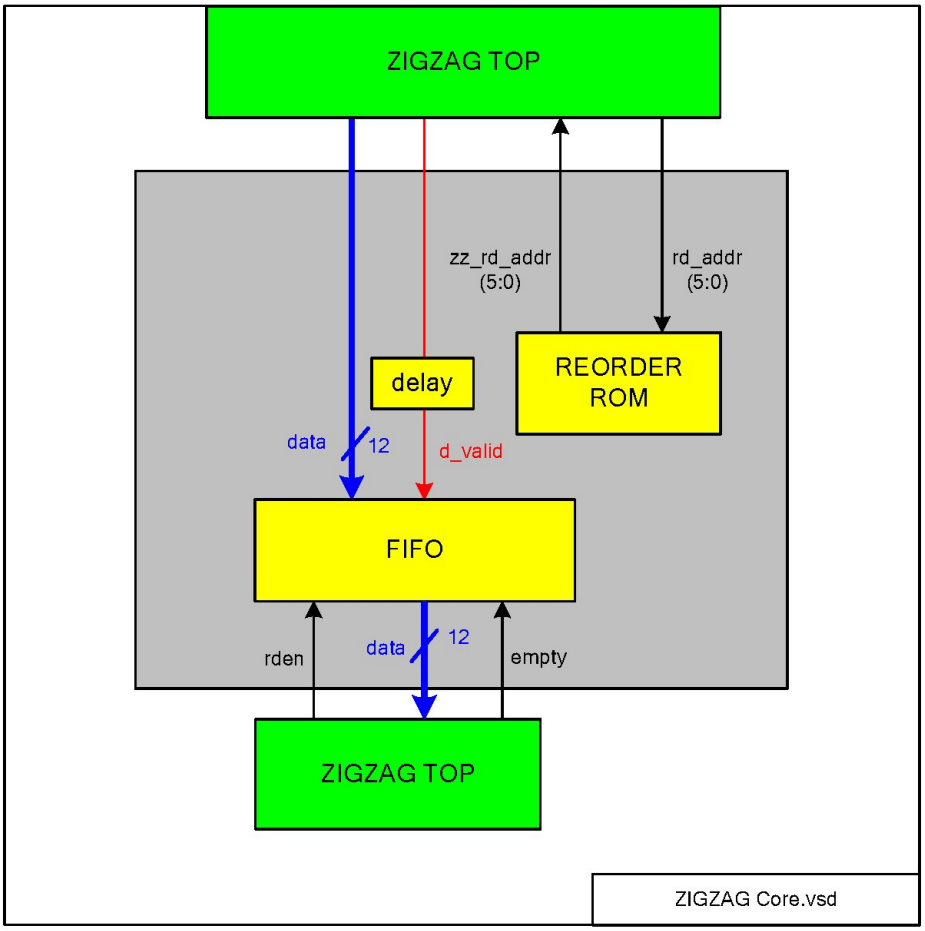

Figure 12

ZigZag Core is performing reordering of input samples according to zig-zag sequence. For that purpose REORDER ROM is used.
This ROM should be filled with following values written from address 0 to address 63:

0,1,8,16,9,2,3,10,
17,24,32,25,18,11,4,5,
12,19,26,33,40,48,41,34,
27,20,13,6,7,14,21,28,
35,42,49,56,57,50,43,36,
29,22,15,23,30,37,44,51,
58,59,52,45,38,31,39,46,
53,60,61,54,47,55,62,63

Note: This table is reverse from JPEG std spec table.
FIFO is used to store reordered samples for next processing step. FIFO size 64x12 bit.

1.5.2 FIFO ctrl

FIFO CTRL is read controller for FIFO. It reads FIFO output as long as it is not empty. Also it generates data valid signal which is simply delayed FIFO read req.

1.5.3 DBUF

Dual port RAM size 2 x 64 x 12 bits.

1.6 QUANTIZER

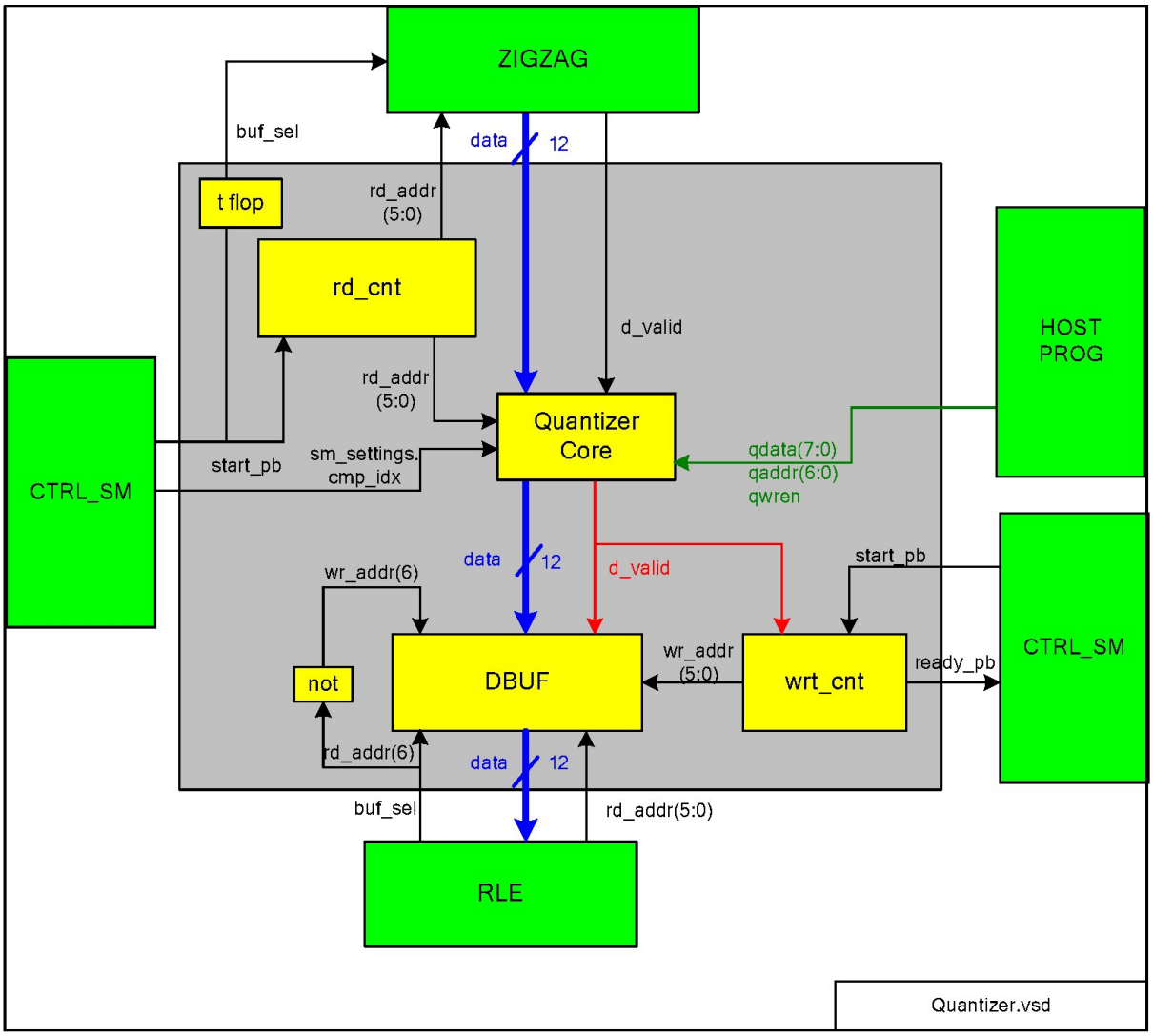

Figure 13

Quantizer performs division of input samples by defined quantization values. Works sample wise. Host can fill in 64 different quantization coefficients to internal RAM (64x8 bits)

Following equation is implemented:

$$FOUT(u, v) = round(FINPUT(u,v)/Q(u,v))$$

- u,v –rectangular coordinates
- FINPUT – input sample to quantizer
- FOUTPUT – output sample from quantizer.
- Rounding to nearest integer

Default quantization table: All ones, i.e. no quantization at all.

When start_pb asserts:

- rd_cnt is reset to 0. 64 samples are then read from ZIGZAG
- wr_cnt is reset
- buf_sel to ZIGZAG is toggled

1.6.1.1 Quantizer Core

Quantizer Core contains Divider and Quantizer RAM. Incoming data is divided by divisor taken from Quantizer RAM. Result is divisor (quantized value).

1.6.2 DBUF

Dual port RAM size 2 x 64 x 12 bits.

1.6.2.1 Divider

Divider for quantizer uses precomputed reciprocals stored in ROM at 16 bits accuracy. Thus input dividend is actually multiplied with reciprocal of divisor to obtain quotient. Quotient is then rounded to nearest integer. No remainder is produced and used.

- Reciprocal ROM has 256 elements where one element is 16 bit number.
- Reciprocals of 8 bit numbers from 0 to 255 are stored in this ROM.
- Since reciprocals are almost all fractional numbers and internal arithmetic is integer, all reciprocal are multiplied by constant factor CK=256*256
- This gives enough accuracy for our purpose
- After multiplication dividend with reciprocal only 12 MSBs of result are taken to final quotient. 16 LSBs are thrown away.
- Bit 15 of multiplier result is, if thinking in fractional numbers first digit after decimal dot. This is good indication if rounding is necessary. Number >=0.5 is rounded to 1.0 and number <0.5 is rounded to 0.0.
- During rounding special care must be taken for negative dividend. For negative numbers to round up 1 must be subtracted from quotient. For positive numbers rounding up 1 must be added to quotient. Mux1 is used for that purpose.
- Divider is quite fast and need only 4 cycles to divider 12 bit signed number by 6 bit unsigned number with rounding included
- Divider is pipelined, after initial latency of 4 cycles new result is given every cycle
- One hardware multiplier is necessary in target architecture

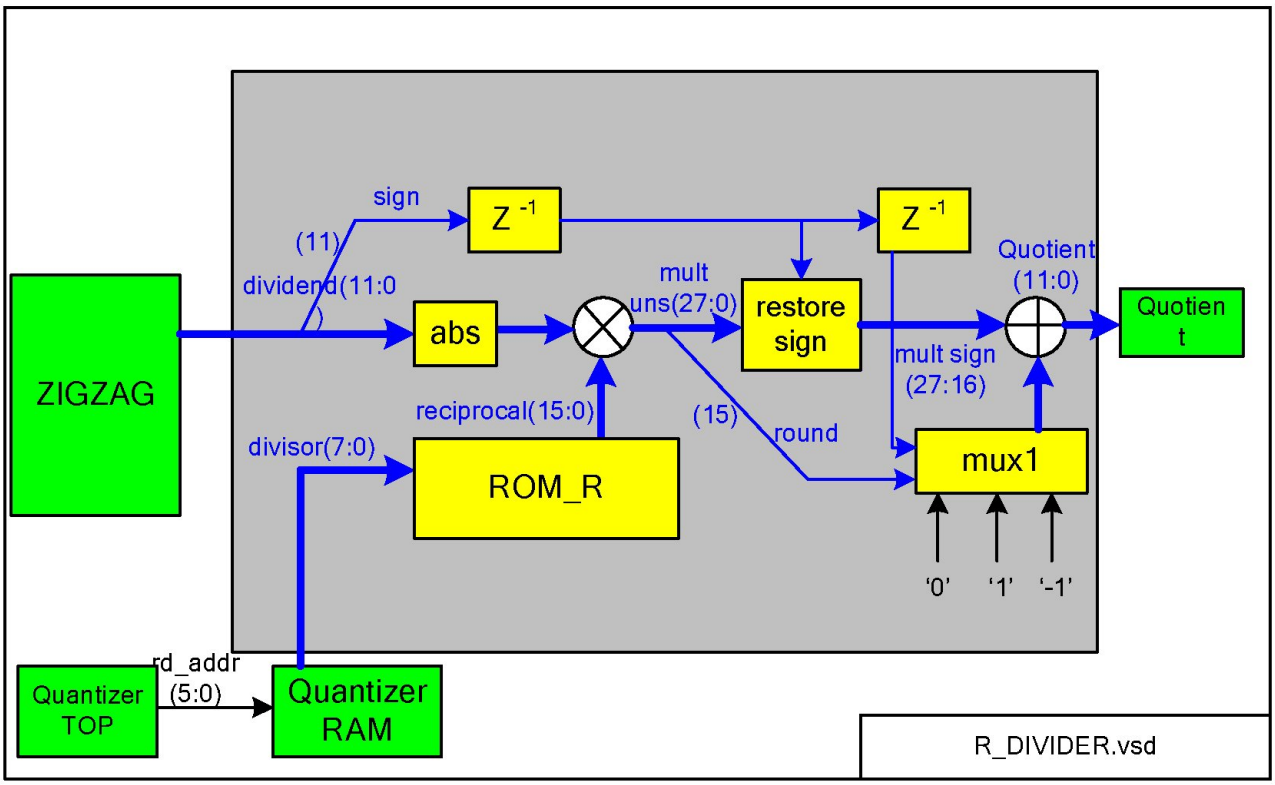

Figure 14

1.6.2.1.1 Mux1

Sign	Round	Mux1 out
0	0	0
0	1	1
1	0	0
1	1	-1

Mux1 is used for rounding result to nearest integer according to table above.

1.6.2.1.2 ABS

Performs arithmetic absolute value. Pass through for positive numbers, two's complement inversion for negative numbers.

1.6.2.1.3 ROM_R

- ROM_R is ROM memory holding reciprocal values for all possible divisor values (1..255).
- For address X, data stored under this address is CK/X where CK is precision constant arbitrary set to 65535.
- For special case address 0, 0 is also stored as value. However this location will never be used as quantization value cannot be zero!

1.6.2.1.4 Restore Sign

Block used to simply restore sign after division is complete. Uses delayed sign bit from dividend to invert its input given that sign bit is '1'. Sign bit is simply MSB of dividend.

1.6.2.2 Quantizer RAM

- Quantizer RAM is actually two RAMs one for luminance, other for Chrominance.
- This could be implemented as one physical RAM 2x64x8 bits.
- Lower half used for luminance (MSB of addr = 0)
- Upper part used for chrominance (MSB of addr = 1)
- address width 7 bits. MSB used for sub-table select.
- data width 8 bits
- quantization values are unsigned integers
- When Y1, Y2 components are processed Luminance table is used
- When Cr, Cb components are processed chrominance table is used
- To select proper quantization sub-table, cmp_idx signal from CtrlSM shall be used. When cmp_idx = 0,1 use luminance, when cmp_idx = 2 or 3 use chrominance sub-table.

1.7 RLE

RLE core performs run-length encoding of data. For input block 8x8 samples, multiple symbol outputs are created consisting of RUNLENGTH, SIZE and AMPLITUDE.

- RUNLENGTH is the number of consecutive zero-valued AC coefficients in the zig-zag sequence preceding the nonzero AC coefficient being represented. For DC coefficient RUNLENGTH is always zero.
- SIZE is the number of bits used to encode AMPLITUDE. SIZE value is between 1 and 10 for AC coefficient. SIZE value is between 1 and 11 for DC coefficient.
- AMPLITUDE is two complement's signed integer. Value ranges from -2047 to 2047 for DC coefficient and from -1023 to 1023 for AC coefficient.

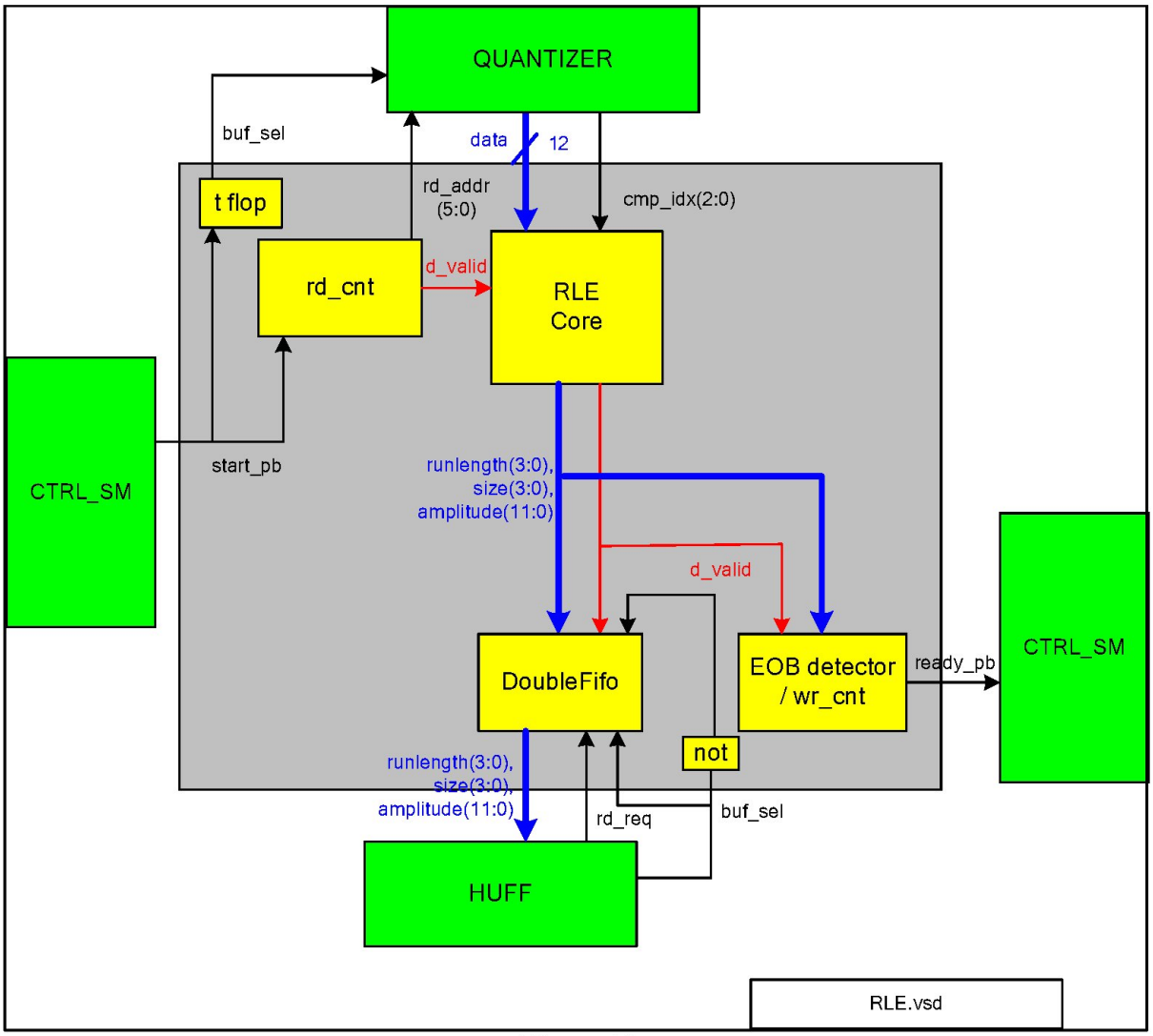

Figure 15

When start_pb asserts:

- buf_sel used by ZIGZAG toggles
- rd_cnt is reset to 0 and starts counting 0..63. -> 64 values are read from QUANTIZER.
- wr_cnt is reset to 0.
- RLE encoding is done on 8x8 block
- RLE encoded data is written to DoubleFIFO

1.7.1 RLE Core

RLE Core perform encoding of 64 input samples to variable number of output symbols. Input data to RLE consists of 64 words. First word is DC coefficient, words 1..63 are AC coefficients

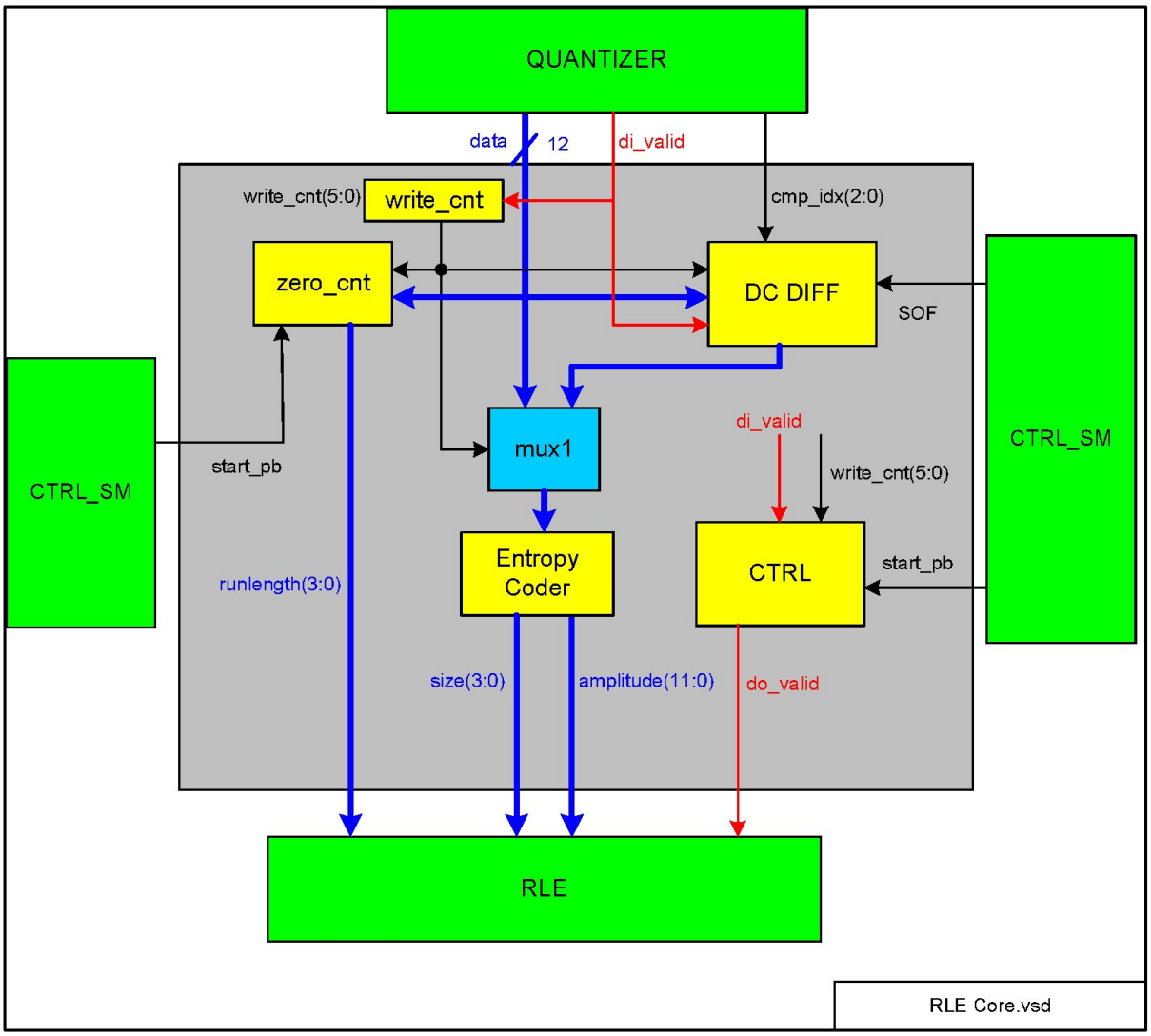

Figure 16

- zero_cnt(3:0) – counts number of zeros preceding non-zero AC coefficient. When start_pb asserts zero_cnt is reset to 0. Output of zero_cnt is used as RUNLENGTH.
- DC DIFF – performs differential encoding of DC coefficient. Also latches current DC value to register to use in next 8x8 block. This latch is reset when SOF asserts.
- MUX1 – switches between DC and AC path depending on write_cnt value. When write_cnt == 0, DC path is used, AC path used otherwise.
- CTRL – controls encoding flow and data valid generation.
- write_cnt is 6 bit counter incremented every time divalid is high. Reset on start_pb.

1.7.1.1 Encoding of input coefficients to output symbols

- DC coefficient (write_cnt == 0) is differentially encoded with DC coefficient from previous 8x8 block for the same color component. That is, RLE_DC = DC(current_block of component x) – DC(last block of component x). To distinguish which component is currently handled sm_settings.cmp_idx input is used.

- On each NON-ZERO input AC coefficient:
  - if zero_cnt <= 15, exactly one output symbol is generated with RUNLENGTH=zero_cnt(3:0). Zero_cnt is then reset to 0. Output data valid asserts for 1 cycle.
  - if zero_cnt > 15 then:
    - generate ZRL extension symbol with RUNLENGTH=15, SIZE=0, AMPLITUDE=0. Output data valid asserts for 1 cycle.
    - set zero_cnt = zero_cnt – 16
    - stall input reading until zero_cnt <= 15
    - repeate procedure "if zero_cnt > 15"
    - if zero_cnt <= 15: one output symbol is generated with RUNLENGTH=zero_cnt. Zero_cnt is then reset to 0. Output data valid asserts for 1 cycle. Input reading can continue.

- On each input AC coefficient equal to ZERO:
  - and AC coefficient is not last in 8x8 block: no output symbol is generated, zero_cnt(5:0) is incremented by 1.
  - and AC coefficient is last in 8x8 block (write_cnt == 63) EOB output symbol is generated: RUNLENGTH=0, SIZE=0, AMPLITUDE=0. Output data valid asserts for 1 cycle.

1.7.2 Entropy Coder

Encodes data sample into pair of symbols: AMPLITUDE (encoded as 2's complement signed integer) and SIZE according to following table:

SIZE	AMPLITUDE
1	-1,1
2	-3,-2,2,3
3	-7..-4,4..7
4	-15..-8,8..15
5	-31..-16,16..31
6	-63..-32,32..63
7	-127..-64,64..127
8	-255..-128,128..255
9	-511..-256,256..511
10	-1023..-512,512..1023
11	–2 047..–1 024,1 024..2 047

Note: Encoded amplitude MSB bit is '1' for positive numbers and '0' for negative numbers!

1.7.3 Read counter

rd_cnt counting is triggered by start_pb assertion.

1.7.4 EOB Detector / Write Counter

- RLE encoding returns variable number of words between 2..63. Final word in case of zero run is called EOB (End Of Block) which is simply runlength=size=amplitude=0 and sample is not DC! EOB detects this and asserts ready_pb pulse.
- Moreover wr_cnt(6:0) is incremented by (1+RUNLENGTH) on every d_valid signal incoming to EOB block. Counter is necessary when EOB code is bypassed (If the last coefficient is not zero).
- assumption: there is always single DC symbol at the beginning. Shortest combination: DC + EOB = 2 symbols.
- It can happen that there are zero runs but no EOB. wr_cnt must adjust its value basing on any zero runs encountered (accumulate RUNLENGTHs) so that when all words are handled counter value = 64 and ready_pb pulse can be asserted.
- Zero run extension. If RUNLENGTH=15 and SIZE=0 this means special Zero Extension symbol was encountered which indicates zero run of 16. In this case write counter must be incremented by 16.

1.7.5 Double FIFO

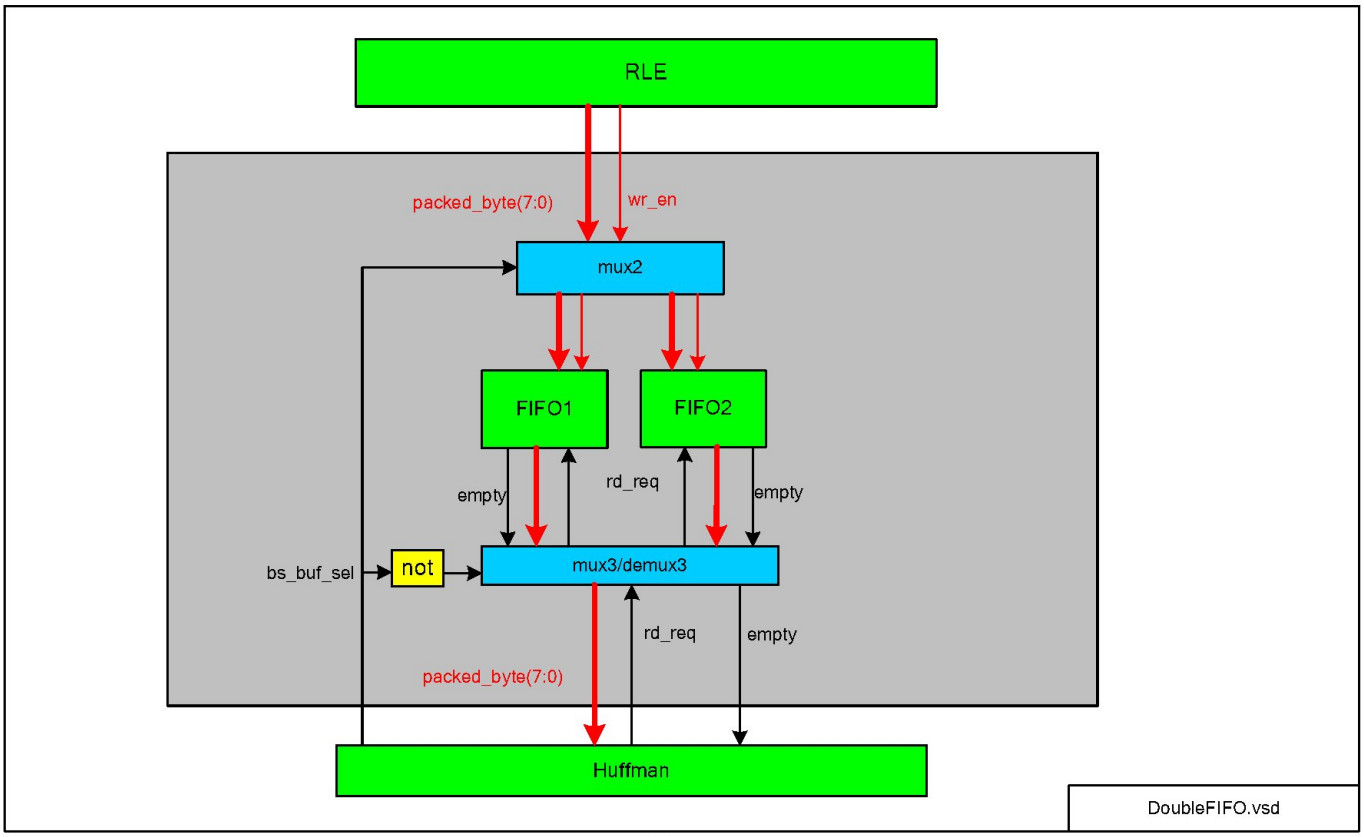

Figure 17

Huffman encoded data are stored to output FIFO. Two FIFOs with size is 64x20 bits each.

Double FIFO block is simply double buffer using two FIFOs.

FIFO is used instead of RAM because RLE encoder can write variable number of encoded words to FIFO thus in this case FIFO implementation is easier than using RAM.

data content in DoubleFIFO:

Q(19:16)	Q(15:12)	Q(11:0)

RUNLENGTH	SIZE	AMPLITUDE

1.8 HUFFMAN Encoder

1.8.1 General Description

Huffman block performs Huffman encoding operation. It converts parallel data into serial bit stream. Serial bit stream output is packed into bytes which are stored in output FIFO. Huffman block operates 8x8 block wise.

1.8.2 Operation

- When start_pb asserts processing is started.
- AC/DC ROMs are used to translate RUNLENGTH/VLI size into variable length code (VLC).
- Variable Length Processor is main FSM logic governing Huffman encoding process.
- buf_sel alternates when start_pb asserts. It is used for double buffering of RLE output buffer.

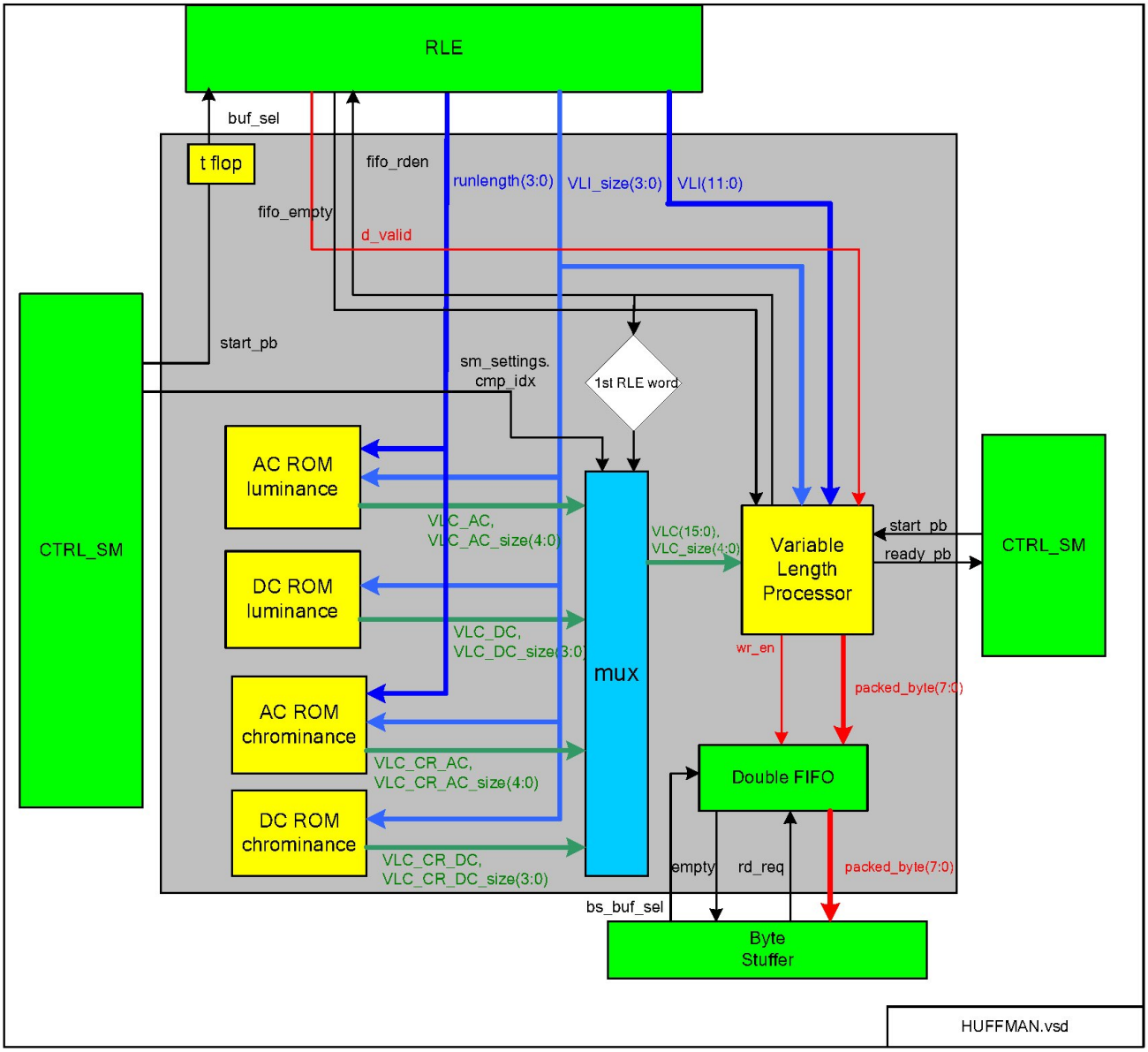

Figure 18

1.8.3 Double FIFO

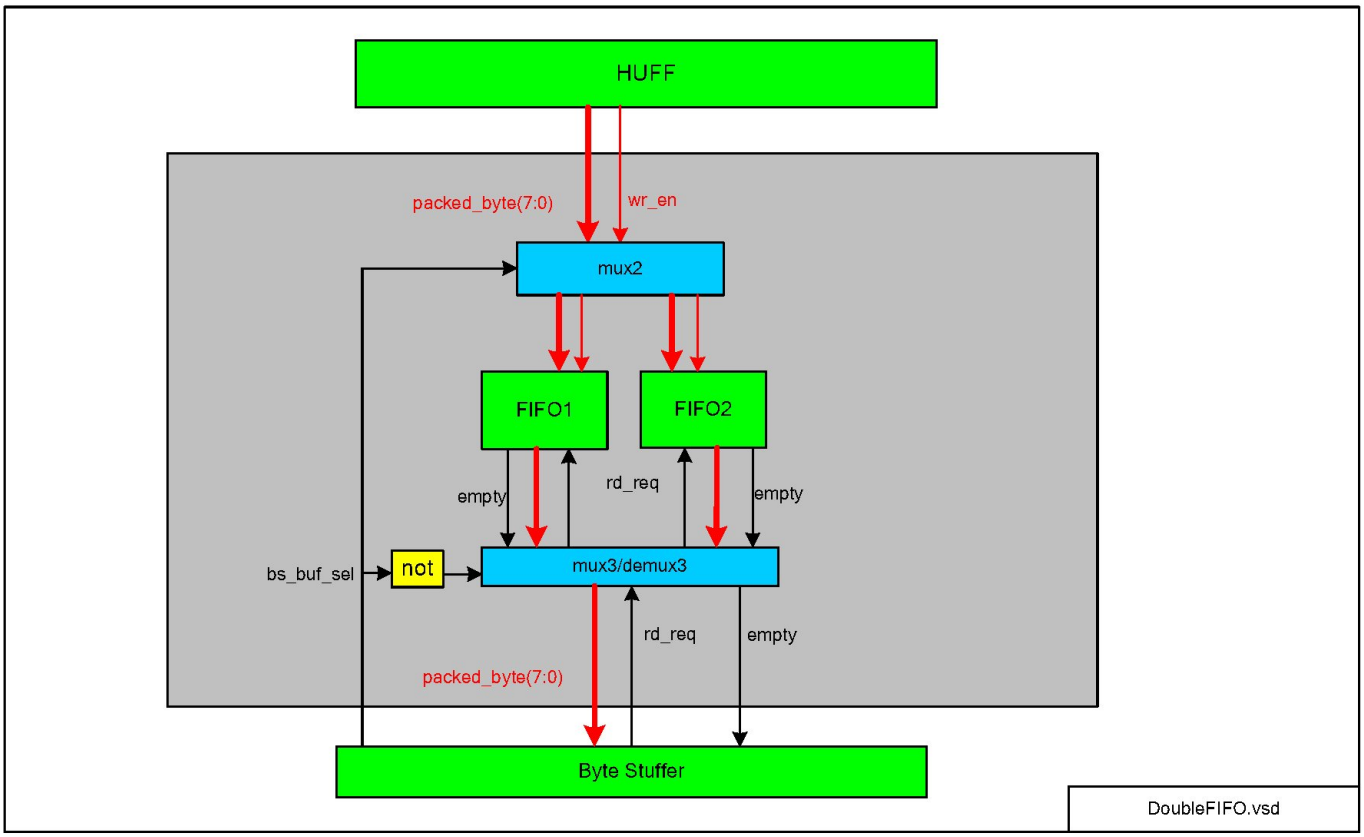

Figure 19

Huffman encoded data are stored to output FIFO. Two FIFOs with size is 2x64x8 bits each.2x 64x8 for assumption that JPEG encoded stream is no more size than 2x of input unencoded stream.

Double FIFO block is simply double buffer using two FIFOs. The idea is that while Huffman writes data for next block Byte Stuffer can read encoded data for previous block.

FIFO is used instead of RAM because Huffman encoder can write variable number of encoded words to FIFO thus FIFO implementation is easier than using RAM.

1.8.4 Mux

Mux basing on component index and input word index decided which ROM output to take as VLC word (luminance/chrominance and AC/DC).

Truth table:

first_rle_word	cmp_idx			
	0	1	2	3
1	VLC1_DC	VLC2_DC	VLC_CR_DC	VLC_CR_DC
0	VLC1_AC	VLC2_AC	VLC_CR_AC	VLC_CR_AC

1.8.5 Variable Length Processor

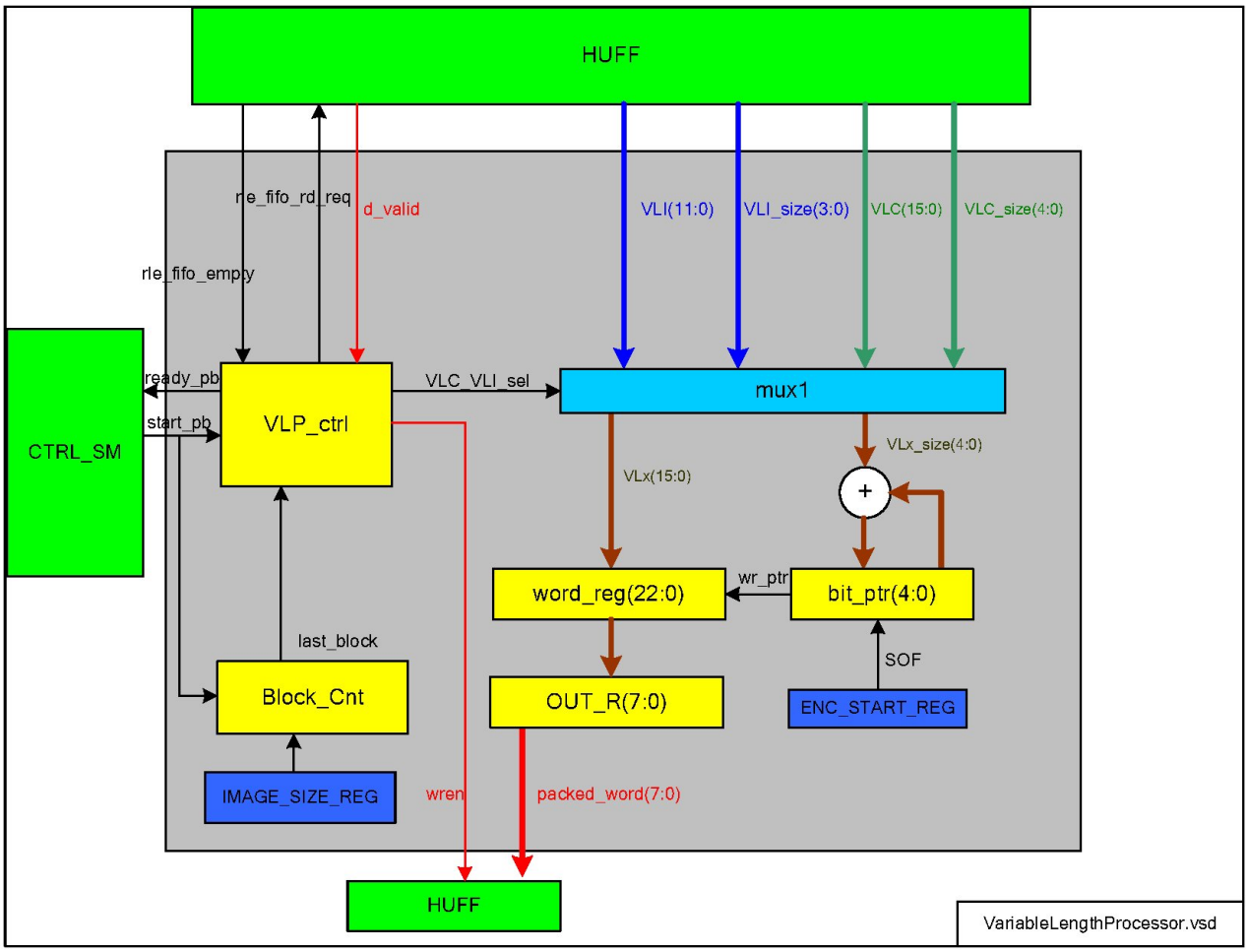

Figure 20

1.8.5.1 General Description

- VLP is Huffman Encoder main state machine and core processing logic. Processing is driven by VLP_ctrl FSM.
- When start_pb asserts all variables are moved into known state and FSM moves to processing state (Run_VLC).
- variable number of words are read from RLE by issuing read_req command until RLE fifo is empty
- Data read from RLE consists of so called RUNLENGTH and VLI (Variable Length Integer code).
- RUNLENGTH (and VLI_SIZE for AC ROM) is then transformed into VLC (Variable Length Code) using luminance and chrominance DC and AC ROMs
- One RLE data symbol can be seen as pair of {VLC,VLI} words.
- DC ROM is used only for first word=0, words 1..63 use AC ROM.
- Since VLI and VLC codes are variable bit length, actual VLC/VLI bit size is given by 2nd argument VLC_size/VLI_size.

1.8.5.2 Serialization

- VLC/VLI encoded pair undergo serialization to bit stream. Serial bit stream is divided into number of output bytes.
- VLC is serialized first, then VLI.
- special bit pointer (bit_ptr(5:0)) is used to track pointer to first empty bit in word register (word_reg)
- VLC/VLI serial bits are mapped to word register (word_reg) starting from bit_ptr position. After that bit_ptr is moved to the right by number of bits in current VLC/VLI word. Bit pointer starts from MSB of word_reg. Bit_ptr=0 is MSB of word_reg.
- After all VLC/VLI words from current block are serialized and stored to output FIFO ready_pb signal asserts

1.8.5.3 data valid

- input data valid d_valid can be simply delayed version of read_req.

1.8.5.4 1st RLE word detection

- set first_rle_word to 1 when start_pb asserts. Reset to zero on first fifo read request to RLE. This way for DC word DC-ROM will be used and for other RLE words AC-ROM will be used.

1.8.5.5 bit pointer

- bit_ptr(4:0) is used to track last used bit in word_reg. Bit_ptr is reset to zero when SOF asserts and is NOT reset when start_pb asserts!.

1.8.5.6 VLP State Machine

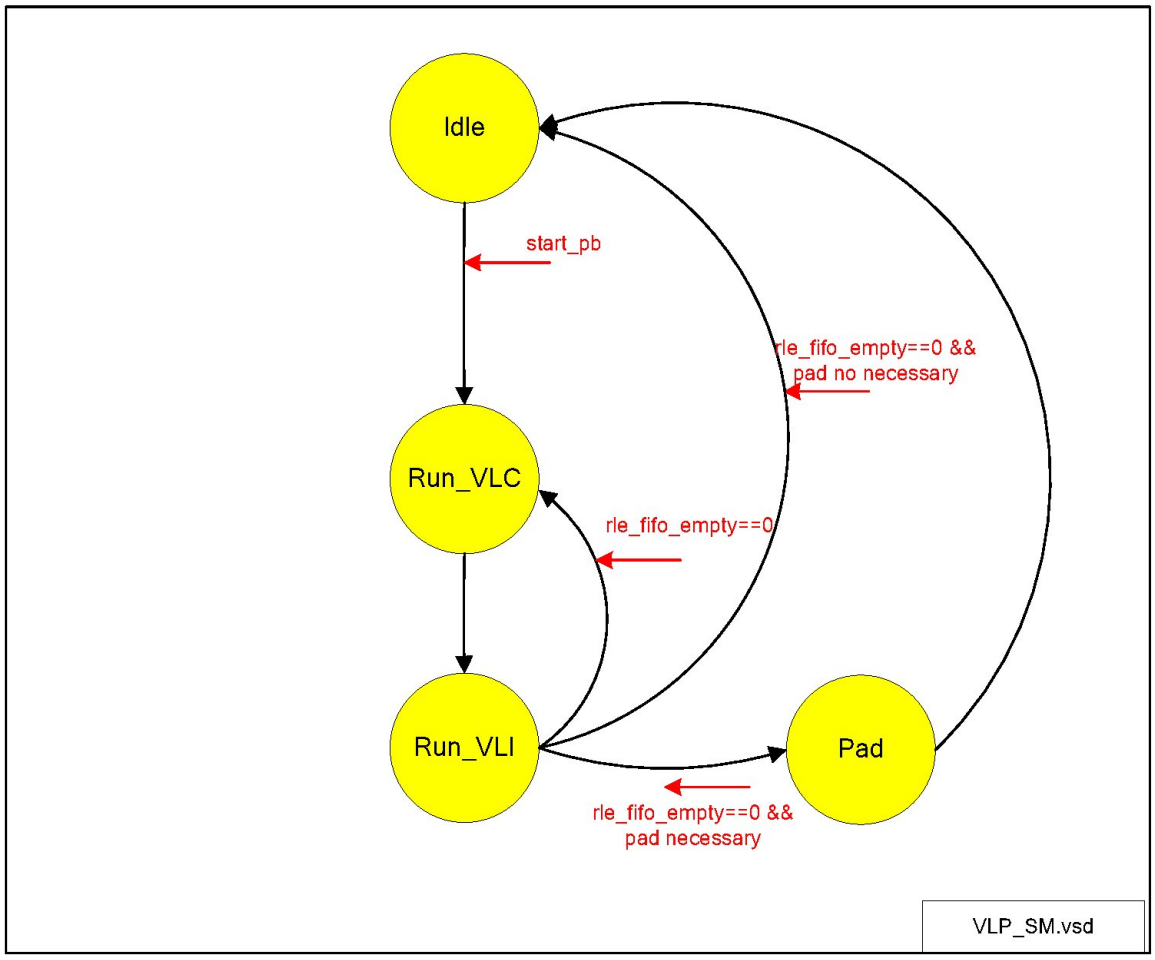

Figure 21

1.8.5.7 VLP SM details

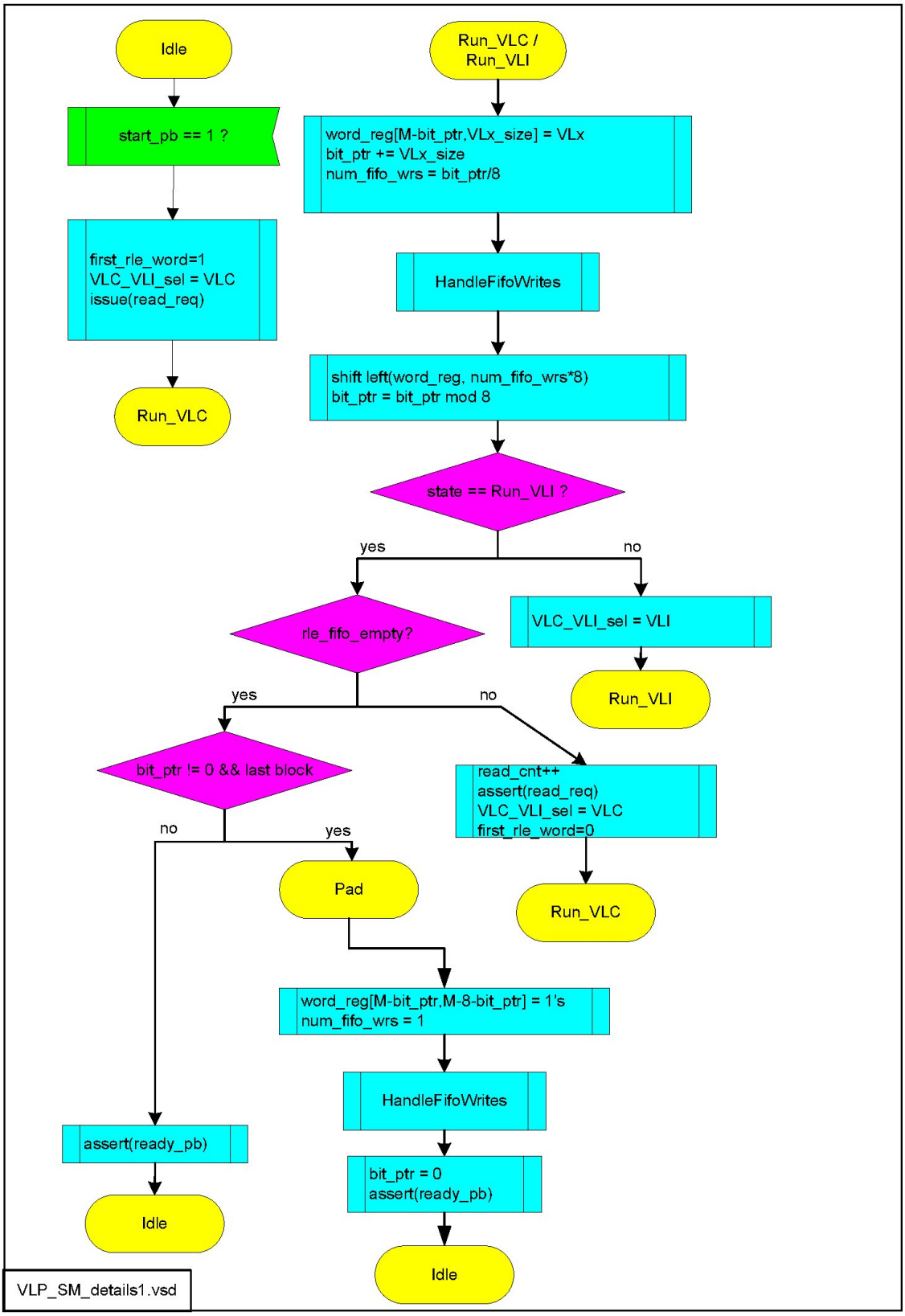

Figure 22

- Idea of above state machine is to write byte-wise Huffman encoded variable length bit stream to output FIFO

- It is assumed that single VLC or VLI serial “word” is no more than 16 bits and no less than 1 bit. 16 comes from: max VLC = max(max VLC DC=9, max VLC AC=16) = 16 bits. Max VLI is is 11 bits. Thus max VLx is 16 bits.
- Entropy-coded segments must be made an integer number of bytes. This is performed as follows: for Huffman coding, 1-bits are used, if necessary, to pad the end of the compressed data to complete the final byte of a segment.
- This means HUFFMAN block needs to know when last 8x8 block is being processed and then pad final byte with ones. Otherwise few last encoded bits (<8) would go lost.
- Last Block detection can be done by dividing known size of image (from programming register). Last Block = number_of_components * (image_width * image_height) / 64
- M = word_reg register bit size-1. Currently M=23-1=22.

1.8.5.7.1 Last_Block / Block_Cnt

It is necessary to know when last 8x8 block occurs to make entropy coded segment integer number of bytes. Last block detector works as follows:

- Block_Cnt is incremented every time start_pb asserts. Its maximum value = 3*(Max_image_width * max_image_height) / 64 = 65536 * 65536 / 64 = 201326592
- Thus bitwidth is ceil(log2(67108864)) = 28 bits necessary
- When Block_Cnt reaches last block [(Block_Cnt == number_of_components*(image_width * image_height)/64] it signals it to VLP by asserting last_block signal.
- Block_Cnt is reset to 0 when SOF asserts

1.8.5.7.2 Word_Reg

- word_reg is intermediate processing register designed to be able to hold code word of maximum size
- Variable Length Code is stored to word_reg MSB first
- word_reg register is used to take incoming data. Its size must be 16+7=23 bits to handle worst case. (7 bits left from previous loop iteration and 16 bits from current loop iteration).

1.8.5.7.3 Procedure HandleFifoWrites

- This procedure is responsible to write output bytes to FIFO. As arguments it takes word_reg (data) and num_fifo_wrs (number of bytes to be written).
- number of bytes to write from single RLE word can vary between 0 and 2 per one VLx.
- Bytes to be written are taken starting from MSB of word_reg.

1.8.6 DC Luminance ROM

DC Luminance ROM is used to perform variable length encoding for 1st sample in single processing block (8x8 samples) for luminance components. For given input VLI_size value, VLC code is returned along with its bit width. VLI_size(3:0) is used as address to DC-ROM.

As per **ITU-T81 Table K.3** – Table for luminance DC coefficient differences

Category	Code length	Code word
0	2	00

1	3	010
2	3	011
3	3	100
4	3	101
5	3	110
6	4	1110
7	5	11110
8	6	111110
9	7	1111110
10	8	11111110
11	9	111111110

Category here means VLI_size. Code length (VLC_DC_size) and Code word (VLC_DC) are mapped to “q” output of DC-ROM.
DC-ROM is 12 words x 13 bits.

One entry in DC-ROM is

Q(12:9)	**Q(8:0)**	
VLC_DC_size(3:0)	zero padding(9-variable length)	VLC_DC(variable length)

size of leading zero padding = maximum(Code Length) - VLC_DC_size
From table above: maximum(Code Length)= 9

1.8.7 DC Chrominance ROM

DC Chrominance ROM is used to perform variable length encoding for 1st sample in single processing block (8x8 samples) for chrominance components Cr and Cb. For given input VLI_size value, VLC code is returned along with its bit width. VLI_size(3:0) is used as address to DC-ROM.

As per **ITU-T81 Table K.4** – Table for chrominance DC coefficient differences

Category	**Code length**	**Code word**
0	2	00
1	2	01
2	2	10
3	3	110
4	4	1110
5	5	11110
6	6	111110
7	7	1111110
8	8	11111110
9	9	111111110
10	10	1111111110
11	11	11111111110

Category here means VLI_size. Code length (VLC_CR_DC_size) and Code word (VLC_CR_DC) are mapped to “q” output of DC-ROM.
DC-ROM is 12 words x 15 bits.

One entry in DC-ROM is

Q(14:11)	**Q(10:0)**	
VLC_CR_DC_size(3:0)	zero padding(11-variable length)	VLC_CR_DC(variable length)

size of leading zero padding = maximum(Code Length) - VLC_DC_size
From table above: maximum(Code Length)= 11 for chrominance

1.8.8 AC Luminance ROM

AC-ROM is similar in operation to DC-ROM. Address is constructed from RUNLENGTH(3:0) and VLI_Size(3:0):
AC_ROM_address(7:4) = RUNLENGTH(3:0)
AC_ROM_address(3:0) = VLI_Size(3:0)

From **ITU-T81 Table K.5** – Table for luminance AC coefficients (sheet 1 of 4)

Few entries from Table K.5 given here as an example:

Run/Size	Code length	Code word
0/0 (EOB)	4	1010
0/1	2	00
0/2	2	01
0/3	3	100
...	...	...
F/A	16	1111111111111110

AC-ROM size is 150 words x 21 bits. 150 words comes from Run range 0..15, Size range 0..10.
21 bit width comes from: 5 bits are necessary to store Code Length and 16 bits are necessary to store Code Word (worst case).

Run/Size there means here RUNLENGTH/ VLI_Size. Code Length means VLC_AC_size and Code word means VLC_AC

One entry in AC-ROM is

Q(20:16)	Q(15:0)	
VLC_AC_Size(4:0)	zero padding(16-variable length)	VLC_AC(variable length)

size of leading zero padding = maximum(Code Length) - (VLC_AC_size)
From table 5: maximum(Code Length)= 16 for AC luminance.

1.8.9 AC Chrominance ROM

AC-ROM is similar in operation to DC-ROM. Address is constructed from RUNLENGTH(3:0) and VLI_Size(3:0):
AC_ROM_address(7:4) = RUNLENGTH(3:0)
AC_ROM_address(3:0) = VLI_Size(3:0)

From **ITU-T81 Table K.6** – Table for chrominance AC coefficients (sheet 1 of 4)

Few entries from Table K.6 given here as an example:

Run/Size	Code length	Code word
0/0 (EOB)	2	00
0/1	2	01

0/2	3	100
0/3	4	1010
...	...	...
F/A	16	1111111111111110

AC-ROM Chrominance size is 150 words x 21 bits. 150 words comes from Run range 0..15, Size range 0..10.
21 bit width comes from: 5 bits are necessary to store Code Length and 16 bits are necessary to store Code Word (worst case).

Run/Size there means here RUNLENGTH/ VLI_Size. Code Length means VLC_AC_size and Code word means VLC_AC

One entry in AC-ROM is

Q(20:16)	**Q(15:0)**	
VLC_AC_Size(4:0)	zero padding(16-variable length)	VLC_CR_AC(variable length)

size of leading zero padding = maximum(Code Length) - (VLC_CR_AC_size)
From table 5: maximum(Code Length)= 16 for chrominance

1.9 Byte Stuffer

In order to ensure that a false marker does not occur within an entropy-coded segment, any X'FF' byte (generated by Huffman encoder, or an X'FF' byte that was generated by the padding of 1-bits of final segment) is followed by a "stuffed" zero byte.

Byte Stuffer stuffs a zero byte whenever the addition of the carry to the data already in the entropy-coded segments creates a X'FF' byte.

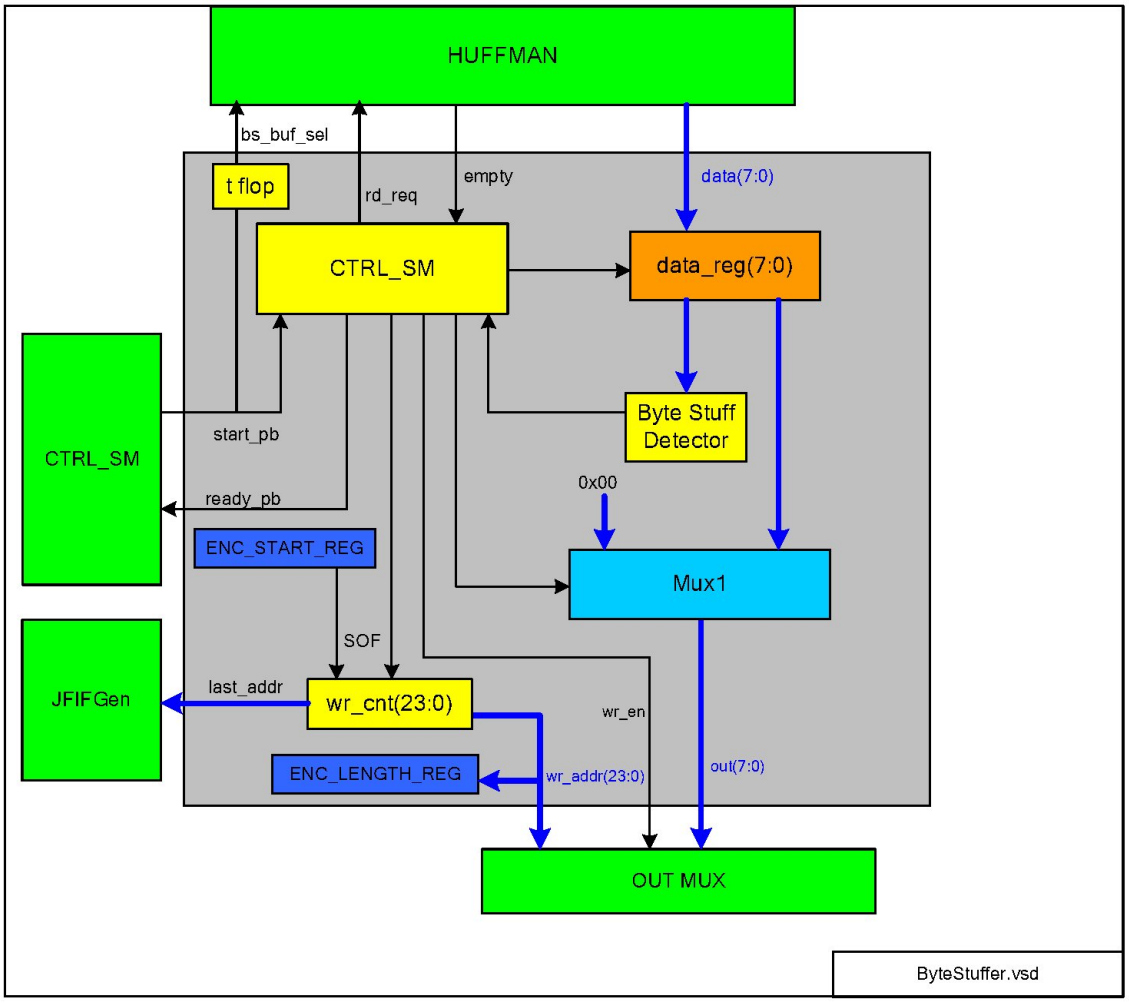

Figure 23

1.9.1 CTRL_SM

Control state machine operates as follows:

- When start_pb asserts Huffman output FIFO is started to being read until it becomes empty
- Every byte read from Huffman FIFO is checked against 0xFF value – this done by Byte Stuff Detector. If 0xFF byte is encountered, byte stuffing with value 0x00 must be performed
- For example: if following bytes are read from Huffman FIFO: AA BB CC FF 22 33, then after byte stuffing there is: AA BB CC FF 00 22 33

- When stuffing is necessary Mux1 is switched to 0x00 path after 0xFF from data path was written to output RAM. Thus, for every 0xFF on input 0xFF, 0x00 is written to output. Otherwise for every input one byte is written to output (the same data)
- When FIFO is empty it means all bytes were processed, ready_pb pulse can assert.

1.9.2 Write Counter

- Counter wr_cnt is used as address to output RAM
- Its current bitwidth is selected arbitrary to 24 bits however necessary size of coded image vary. Thus bitwidth with quite big margin was selected.
- wrt_cnt is preset when SOF asserts to JFIF header size. This is to put coded data byte stream after the JFIF header.
- wrt_cnt is incremented on every new byte written to output RAM (wren – write enable)
- Number of coded bytes stored to RAM (including header) can be read out by Host from ENC_LENGTH_REG. It is taken from wr_addr + 2. 2 comes from EOI marker attached later in JFIF Header Generator.

1.9.3 Byte Stuff Detector

- Simple comparator which checks if input byte is equal to 0xFF.

1.9.4 bs_buf_sel

Double buffer selector. Flips on every start_pb assertion.

1.9.5 last_addr

After all encoded bytes are written, EOI (End Of Image) marker must be written. JFIFGen which does that must know last address written by ByteStuffer to append EOI right after it.

1.10 JFIF Header Generator

Default JFIF header is generated from onchip RAM.

Most of header fields are hardcoded, however some are configurable and must be programmed by Host.

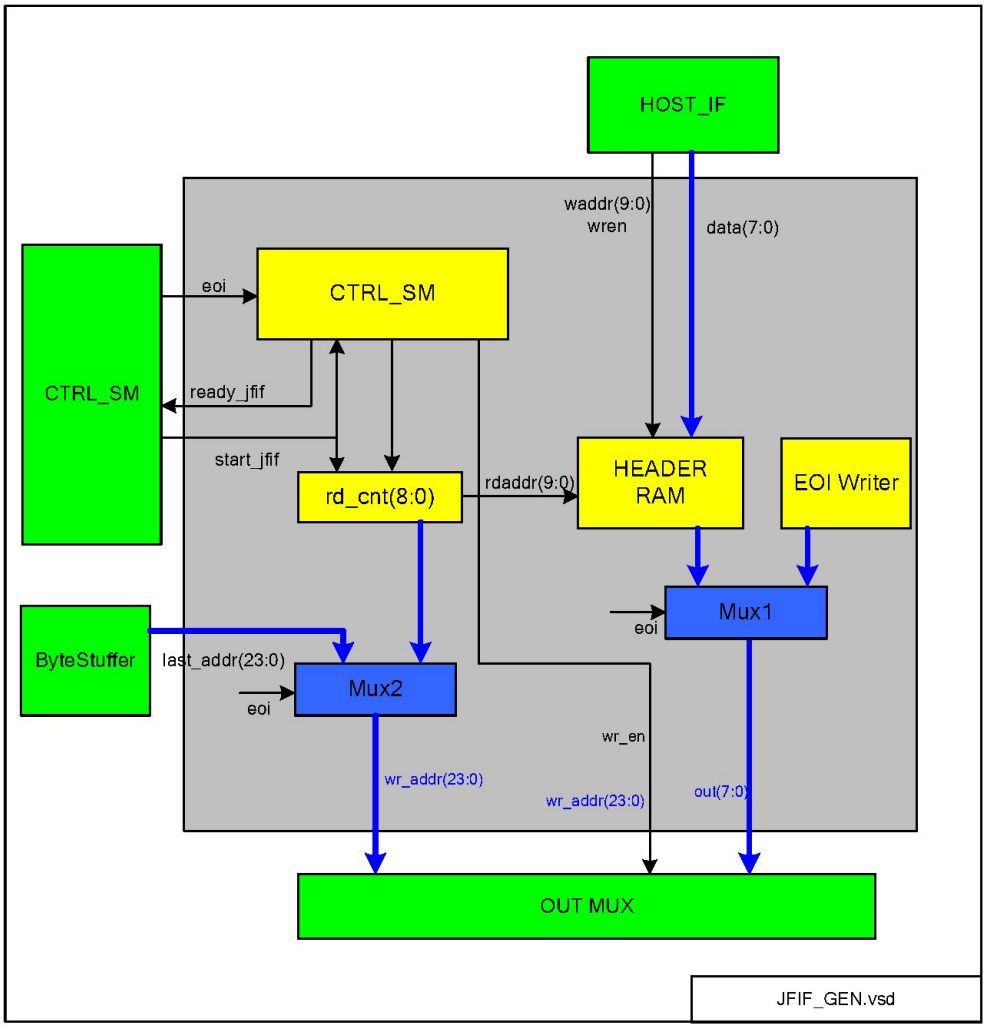

Figure 24

These are:

SIZE_Y_H, SIZE_Y_L – upper and lower byte of image height
SIZE_X_H, SIZE_X_L – upper and lower byte of image width
Number of components – 0=grayscale, 3=color. Fixed to 3 (RGB).
QL0....QL63 – Luminance Quantization table in zig zag order
QC0....QC63 – Chrominance Quantization table in zig zag order

1.10.1 Header RAM Programming

When Host performs programming of JPEG coding parameters, some of them are also passed to JFIF generator module and are written to Header RAM:

- Image Height and Width: written when IMAGE_SIZE_REG is written by Host. Write addresses decimal 25, 26 for Height, 27, 28 for Width.
- Number of components: Fixed to 03! Programming not supported, always 3 components used.
- Luminance Quantization table written from offset 44 decimal.
- Chrominance Quantization table written from offset 44+69=113 decimal.

1.10.2 JFIF Generator

When start_jfif asserts and eoi = 0:

- rd_cnt is reset. rd_cnt then counts from 0..622 (JFIF header size=623 bytes). This reads Header RAM whose output is then saved to Output RAM.
- After all bytes from Header RAM are transmitted ready_jfif asserts.

When start_jfif asserts and eoi = 1:

- EOI marker “FF D9” is written after coded byte stream to “close” image.

1.10.3 EOI Writer

Writes two EOI marker bytes “FF D9” to signal end of image.

1.10.4 Mux1/Mux2

Switch path depending on eoi signal from JPEG CtrlSM. When eoi is high EOI marker is written, otherwise JFIF header is written to output.

1.10.5 Header RAM

Size 1024x8 bits. Dual port RAM. Most of the content taken from HEX initialization file. Some fields are configured by Host IF.

1.10.6 Generic Header

Generic header defaults are stored in RAM using preinitialization. These defaults are (stored starting from address 0):

Description	Data (hex byte)
SOI, start of image marker	FF, D8
JFIF header	
Marker APP0	FF, E0
Length (16 bytes)	00, 10
“JFIF” ascii zero terminated	4A, 46, 49, 46, 00
Version	01, 01
Units	00
XDensity	00, 01
YDensity	00, 01
XThumbnail (no thumbnail)	00
YThumbnail (no thumbnail)	00

Start Of Frame	
Baseline DCT, SOF0 marker	FF, C0
Length (11 bytes)	00, 11
Sample Precision	08
Height	SIZE_Y_H, SIZE_Y_L
Width	SIZE_X_H, SIZE_X_L
Number of components (grayscale/color)	03
Scan 1: 1:1 horiz, (byte) 1:1 vertical, (byte) use QT 0	01, 21, 00
Scan 2: 2:1 horiz, (byte) 1:1 vertical, (byte) use QT 1	02, 11, 01
Scan 3: 2:1 horiz, (byte) 1:1 vertical, (byte) use QT 1	03, 11, 01
Define Quantization table (Luminance)	
marker	FF, DB
Length (67 bytes)	00, 43
8 bit values, table 0	00
quantization 8-bit values (64 entries, zigzag order)	QL0....QL63
Define Quantization table (Chrominance)	
marker	FF, DB
Length (67 bytes)	00, 43
8 bit values, table 1	01
quantization 8-bit values (64 entries, zigzag order)	QC0....QC63
Define Huffman Table (Luminance)	
Marker	FF, C4
Length (31 bytes)	00, 1F
DC, Table 0	00
L1..L16	00, 01, 05, 01, 01, 01, 01, 01 01, 00, 00, 00, 00, 00, 00, 00
Symbol codes for L1..L16 (Vij)	00, 01, 02, 03, 04, 05, 06, 07, 08, 09, 0A, 0B
Define Huffman Table (Chrominance)	
Marker	FF, C4
Length (31 bytes)	00, 1F
DC, Table 1	01
L1..L16	00 03 01 01 01 01 01 01 01 01 01 00 00 00 00 00
Symbol codes for L1..L16 (Vij)	00 01 02 03 04 05 06 07 08 09 0A 0B
Define Huffman Table (Luminance)	
Marker	FF, C4
Length (181 bytes)	00, B5
AC, table 0	10
L1..L16	00, 02, 01, 03, 03, 02, 04, 03, 05, 05, 04, 04, 00, 00, 01,
Symbol codes for L1..L16 (Vij)	7D, 01, 02, 03, 00, 04, 11, 05, 12, 21, 31, 41, 06, 13, 51, 61, 07, 22, 71, 14, 32, 81, 91, A1, 08, 23, 42, B1, C1, 15, 52, D1, F0, 24, 33, 62, 72, 82, 09, 0A,

	16, 17, 18, 19, 1A, 25, 26, 27, 28, 29, 2A, 34, 35, 36, 37, 38, 39, 3A, 43, 44, 45, 46, 47, 48, 49, 4A, 53, 54, 55, 56, 57, 58, 59, 5A, 63, 64, 65, 66, 67, 68, 69, 6A, 73, 74, 75, 76, 77, 78, 79, 7A, 83, 84, 85, 86, 87, 88, 89, 8A, 92, 93, 94, 95, 96, 97, 98, 99, 9A, A2, A3, A4, A5, A6, A7, A8, A9, AA, B2, B3, B4, B5, B6, B7, B8, B9, BA, C2, C3, C4, C5, C6, C7, C8, C9, CA, D2, D3, D4, D5, D6, D7, D8, D9, DA, E1, E2, E3, E4, E5, E6, E7, E8, E9, EA, F1, F2, F3, F4, F5, F6, F7, F8, F9, FA
Define Huffman Table (Chrominance)	
Marker	FF, C4
Length (181 bytes)	00, B5
AC, table 1	11
L1..L16	00 02 01 02 04 04 03 04 07 05 04 04 00 01 02 77
Symbol codes for L1..L16 (Vij)	00 01 02 03 11 04 05 21 31 06 12 41 51 07 61 71 13 22 32 81 08 14 42 91 A1 B1 C1 09 23 33 52 F0 15 62 72 D1 0A 16 24 34 E1 25 F1 17 18 19 1A 26 27 28 29 2A 35 36 37 38 39 3A 43 44 45 46 47 48 49 4A 53 54 55 56 57 58 59 5A 63 64 65 66 67 68 69 6A 73 74 75 76 77 78 79 7A 82 83 84 85 86 87 88 89 8A 92 93 94 95 96 97 98 99 9A A2 A3 A4 A5 A6 A7 A8 A9 AA B2 B3 B4 B5 B6 B7 B8 B9 BA C2 C3 C4 C5 C6 C7 C8 C9 CA D2 D3 D4 D5 D6 D7 D8 D9 DA E2 E3 E4 E5 E6 E7 E8 E9 EA F2 F3 F4 F5 F6 F7 F8 F9 FA
Start Of Scan	
Marker	FF, DA
Length	00, 0C
Number of Image components (color)	03

Component 1, use DC/AC huff tables 0/0	01, 00
Component 2, use DC/AC huff tables 1/1	02, 11
Component 3, use DC/AC huff tables 1/1	03, 11
Ss	00
Se	3F
Ah, Al	00
Data Scan	
JPEG encoded byte stream goes here ...	
End of Image	
Marker: EOI	FF, D9

NOTE: This header must be pre-programmed to FPGA memory for example using mif or hex files. For ModelSim simulation hex file can be used.

1.11 Programming Interface

1.11.1 General

JPEG core programming is done via OPB Host interface using for example Xilinx Microblaze processor.

Only 32 bit aligned accesses are supported. It means input address must be multiple of 4.

All registers are 32 bits wide. Although some bits may be unused.

R/W = read/write
WO = write only
RO = read only

1.11.2 Core address map

Address	Register	
0x0 0000	ENC_START_REG	WO
0x0 0004	IMAGE_SIZE_REG	R/W
0x0 000C	ENC_STS_REG	RO
0x0 0010	-	-
0x0 0014	ENC_LENGTH_REG	RO
0x00 0100 ... 0x00 01FF	QUANTIZER_RAM_LUM	WO
0x00 0200 ... 0x00 02FF	QUANTIZER_RAM_CHR	WO

1.11.3 Register Descriptions

ENC_START_REG	WO	Default: 0x0
-	D31..D3	-
cmp_max	D2..D1	Number of image components. 3=color RGB input, YUV output. Must write 3 here.
SOF	D0	Start Of Frame. Writing '1' to this bit starts JPEG encoding process.

IMAGE_SIZE_REG	R/W	Default: 0x0
size_x	D31..D16	Image width in pixels (must be multiple of 16)
size_y	D15..D0	Image height in pixels (must be multiple of 8)

ENC_STS_REG	RO	Default: 0x0
-	D31..D2	-
done	D1	JPEG encoding complete
busy	D0	JPEG encoding in progress

ENC_LENGTH_REG	RO	Default: 0x0

Header, EOI and data length included. Final JPEG file size.

-	D31..D10	-
length	D23..D0	coded stream length (in bytes)

QUANTIZER_RAM_LUM	WO	Default: 0x1

64 entries for Luminance table. Address increment every 4!

-	D31..D8	-
q_coeff	D7..D0	Quantization coefficient. Unsigned integer range 1..255

QUANTIZER_RAM_CHR	WO	Default: 0x1

64 entries for Chrominance table. Address increment every 4!

-	D31..D8	-
q_coeff	D7..D0	Quantization coefficient. Unsigned integer range 1..255

1.12 Users Manual

To perform JPEG encoding of image Host must perform steps below:

- Write IMAGE_SIZE_REG with width and height of image to be encoded
- Fill in Quantizer table using QUANTIZER_RAM (luminance and chrominance)
- write SOF bit in ENC_START_REG to start encoding process
- Write raw bitmap image samples BUF_FIFO.
- Host either waits for interrupt ready_int (pulse) or polls for JPEG encoding done flag in ENC_STS_REG to know when encoding is complete
- Host can read ENC_LENGTH_REG to get length of encoded byte stream in output RAM
- Enjoy!

1.13 Simulation

- VHDL simulation can be run by executing sim.do TCL script under ModelSim.
- Input bitmap image is in text format and is called **test.txt** – must be in directory where sim.do is placed
- **test.txt** image text format can be generated using img2txt.m Matlab script
- **img2txt.m** is in **./testbench/matlab/img2txt.** It takes as input **test.bmp** and converts it to **test.txt**
- **test.bmp** is **24 bit** RGB bitmap
- image height must have dimensions **multiple of 8, image width must be multiple of 16**
- Image width must not be higher than **JPEG_PKG.VHD** constant **C_MAX_LINE_WIDTH** otherwise simulation error will occur. In this case increase this constant to support bigger input image width. Be warned that bigger this constant value is, the more FPGA memory will be needed when core is synthesized.
- JPEG encoded output byte stream is saved to file **OUT_RAM.txt** as hex byte string and also to **test_out.jpg** as regular JPEG format file.

1.13.1 OPB Master BFM

- OPB Master Bus Functional Model is used to drive OPB slave bus of EV_JPEG_ENC core to configure JPEG Core and also it loads image through fast FIFO-like interface.
- This BFM is in file **HostBFM.vhd**
- Quantization table can be modified in HostBFM.vhd file by changing **qrom** constant. There are few predefined quantization tables, just comment all of them except one to use 100%, 75% or 50%.