

Cristian Gil Morales

# CCSDS-123 Issues 1 & 2 Implementation on FPGA

Master's thesis in Electronic Systems Design  
Supervisor: Milica Orlandic  
January 2022



Cristian Gil Morales

# **CCSDS-123 Issues 1 & 2 Implementation on FPGA**

Master's thesis in Electronic Systems Design  
Supervisor: Milica Orlandic  
January 2022

Norwegian University of Science and Technology  
Faculty of Information Technology and Electrical Engineering  
Department of Electronic Systems





*Dedico mi mayor logro académico a mi aún amada Carolina.  
Este ha sido el último paso de un camino que jamás debí haber tomado.*



## Abstract

The CCSDS-123 standard is a low-complex but efficient prediction-based (de)compression algorithm for multispectral and hyperspectral imagers and sounders, that provides a compacted data transmission via a communication link [11].

The present report proposes a full implementation of the Issues 1 & 2 algorithms in VHDL-2008, with a fully-configurable nature by means of different *package* files. The Issue 2 revision extends the capabilities of Issue 1 (but still backwards compatible), providing the possibility to perform either lossless or near-lossless compression of the image data [10].

Because of the performance and timing requirements that a space-related application demands, this implementation must be accelerated by hardware, in this case a FPGA. FPGAs are increasingly becoming the most suitable platforms in terms of performance, energy efficiency and reconfigurability [13].

This paper covers every single aspect of the CCSDS-123 Issues 1 & 2 algorithms, except the *Block-Adaptive Entropy Coder* (from Issue 1), so being by far the most complete implementation to run on a FPGA, and the current state-of-the-art.

The presented implementation of (almost) one independent IP module per different operation (exactly in the same way as the provided documentation structures it) ensures a high grade of modularity, extendability, scalability, bug tracking and code readability.

A complete validation by simulation of the system, both by visually tracking all signals and comparing with an already validated partial implementation in VHDL too, is performed with successful results. The design requires around 10% of the hardware resources from the FPGA, and it is currently capable to run at a speed of 40MHz (and with a huge potential to easily accelerate it).

The performance tests define a specific number of total clock cycles to fully compress an image (very closely related to the input image dimensions) and an image compression ratio of between 40% and 60% the original input image size.

I would like to thank the Norwegian University of Science and Technology, for giving me this great opportunity to work with them, and specially to my supervisor Milica Orlandic, who has followed my development and assisted me when needed.

# Contents

<b>Acronyms</b>	<b>14</b>
<b>1 Introduction</b>	<b>16</b>
<b>2 Background</b>	<b>18</b>
2.1 Input image . . . . .	18
2.1.1 Input samples order . . . . .	19
2.2 CCSDS-123 Issue 2 algorithm . . . . .	21
2.3 Predictor block . . . . .	21
2.3.1 Adder . . . . .	22
2.3.2 Quantizer . . . . .	22
2.3.2.1 Fidelity Control . . . . .	23
2.3.2.1.1 Periodic Error Limit Updating . . . . .	24
2.3.3 Mapper . . . . .	24
2.3.3.1 Scaled Difference . . . . .	24
2.3.4 Sample Representative . . . . .	25
2.3.4.1 Clipped Quantizer Bin Center . . . . .	26
2.3.4.2 Double-Resolution Sample Representative . . . . .	26
2.3.5 Prediction . . . . .	26
2.3.5.1 Local Sum . . . . .	28
2.3.5.2 Local Differences . . . . .	30
2.3.5.3 Local Differences Vector . . . . .	31
2.3.5.4 Weight values . . . . .	32
2.3.5.5 Weight Update Scaling Exponent . . . . .	34
2.3.5.6 Double-Resolution Prediction Error . . . . .	34
2.3.5.7 Weights Vector . . . . .	34
2.3.5.8 Predicted Central Local Difference . . . . .	35
2.3.5.9 High-Resolution Predicted Sample . . . . .	35
2.3.5.10 Double-Resolution Predicted Sample . . . . .	36
2.3.5.11 Predicted Sample . . . . .	36
2.4 Encoder block . . . . .	37
2.4.1 Encoder Header . . . . .	38
2.4.1.1 Image Metadata . . . . .	38
2.4.1.1.1 Supplementary Information Tables . . . . .	40
2.4.1.2 Predictor Metadata . . . . .	42
2.4.1.3 Encoder Metadata . . . . .	47
2.4.2 Encoder Body . . . . .	48
2.4.2.1 Sample-Adaptive Entropy Coder . . . . .	49
2.4.2.1.1 Sample-Adaptive Statistic . . . . .	49



2.4.2.1.2	Sample-Adaptive GPO2 Coder . . . . .	50
2.4.2.2	Hybrid Entropy Coder . . . . .	51
2.4.2.2.1	Hybrid Statistic . . . . .	51
2.4.2.2.2	Hybrid High-Entropy Coder . . . . .	52
2.4.2.2.3	Hybrid Low-Entropy Coder . . . . .	53
2.4.2.2.4	Hybrid Compressed Image Tail . . . . .	54
2.4.3	Output packets generation . . . . .	54
2.5	Differences between Issues 1 and 2 . . . . .	55
2.6	VUnit framework . . . . .	57
2.7	Logic Synthesis . . . . .	58
2.8	Implementation . . . . .	58
2.9	HDL considerations . . . . .	59
2.9.1	VHDL signed vs unsigned signals . . . . .	59
2.9.2	Synthesis design constraints . . . . .	60
<b>3</b>	<b>Design</b> . . . . .	<b>61</b>
3.1	Overview . . . . .	61
3.1.1	Timing diagram . . . . .	62
3.2	Development tools . . . . .	63
3.3	Hardware platform . . . . .	63
3.4	Source code architecture . . . . .	65
3.4.1	Packages . . . . .	65
3.4.2	Block diagrams description . . . . .	66
3.5	CCSDS-123-Issue2 Top Entity IP . . . . .	68
3.5.1	Top Entity IP configuration . . . . .	69
3.6	Image Coordinates Control IP . . . . .	71
3.7	Predictor Top IP . . . . .	71
3.7.1	Adder IP . . . . .	73
3.7.2	Quantizer IP . . . . .	74
3.7.2.1	Fidelity Control IP . . . . .	74
3.7.2.1.1	Error Limit Values Table . . . . .	76
3.7.3	Mapper IP . . . . .	77
3.7.3.1	Scaled Difference IP . . . . .	77
3.7.4	Sample Representative IP . . . . .	78
3.7.4.1	Clipped Quantizer Bin Center IP . . . . .	78
3.7.4.2	Double-Resolution Sample Representative IP . . . . .	79
3.7.5	Prediction IP . . . . .	79
3.7.5.1	Samples Store IP . . . . .	81
3.7.5.1.1	Shift Register IP . . . . .	82
3.7.5.2	Local Sum IP . . . . .	83
3.7.5.3	Local Differences IP . . . . .	84
3.7.5.4	Local Differences Vector IP . . . . .	85
3.7.5.5	Weight Update Scaling Exponent IP . . . . .	86
3.7.5.6	Double-Resolution Prediction Error IP . . . . .	86

3.7.5.7	Weights Vector IP . . . . .	86
3.7.5.8	Predicted Central Local Difference IP . . . . .	87
3.7.5.9	High-Resolution Predicted Sample IP . . . . .	88
3.7.5.10	Double-Resolution Predicted Sample IP . . . . .	88
3.7.5.11	Predicted Sample IP . . . . .	88
3.8	Predictor-Encoder Interconnection IP . . . . .	89
3.8.1	Parallel Synchronous FIFOs IP . . . . .	90
3.9	Encoder Top IP . . . . .	91
3.9.1	Encoder Header IP . . . . .	92
3.9.1.1	Image Metadata IP . . . . .	93
3.9.1.1.1	Supplementary Information Tables . . . . .	94
3.9.1.2	Predictor Metadata IP . . . . .	95
3.9.1.3	Encoder Metadata IP . . . . .	95
3.9.2	Encoder Body IP . . . . .	96
3.9.2.1	Sample-Adaptive Entropy Coder IP . . . . .	97
3.9.2.1.1	Sample-Adaptive Statistic IP . . . . .	97
3.9.2.1.2	Sample-Adaptive GPO2 Coder IP . . . . .	98
3.9.2.2	Hybrid Entropy Coder IP . . . . .	100
3.9.2.2.1	Hybrid Statistic IP . . . . .	101
3.9.2.2.2	Hybrid High-Entropy Coder IP . . . . .	102
3.9.2.2.3	Hybrid Low-Entropy Coder IP . . . . .	103
3.9.2.2.4	Hybrid Compressed Image Tail IP . . . . .	105
3.9.3	Packer IP . . . . .	106
<b>4</b>	<b>Validation Plan</b>	<b>109</b>
4.1	Validation scope . . . . .	109
4.2	Validation tools . . . . .	109
4.2.1	VUnit testbenches . . . . .	109
4.2.2	Vivado TCL framework . . . . .	112
4.3	Test-cases . . . . .	114
<b>5</b>	<b>Results</b>	<b>115</b>
5.1	Bitstream generation . . . . .	115
5.2	HW integration reports . . . . .	116
5.2.1	Power consumption report . . . . .	116
5.2.2	Utilization report . . . . .	117
5.2.3	Timing report . . . . .	118
5.3	Functionality outcome . . . . .	119
5.3.1	Image Coordinates Control IP block . . . . .	119
5.3.2	Predictor IP block . . . . .	120
5.3.2.1	Adder IP sub-block . . . . .	120
5.3.2.2	Quantizer IP sub-block . . . . .	120
5.3.2.3	Mapper IP sub-block . . . . .	121
5.3.2.4	Sample Representative IP sub-block . . . . .	121

5.3.2.5	Prediction IP sub-block . . . . .	122
5.3.2.6	Predictor Top IP integration test . . . . .	122
5.3.3	Encoder IP block . . . . .	123
5.3.3.1	Encoder Header IP sub-block . . . . .	123
5.3.3.2	Sample-Adaptive Entropy Coder IP sub-block . . . . .	124
5.3.3.3	Hybrid Entropy Coder IP sub-block . . . . .	124
5.3.3.4	Packer IP sub-block . . . . .	125
5.3.3.5	Encoder Top IP integration test . . . . .	126
5.3.4	Top Entity IP integration test . . . . .	126
5.4	Performance & Final results . . . . .	127
5.5	Time planning . . . . .	128
<b>6</b>	<b>Discussion</b>	<b>130</b>
<b>7</b>	<b>Related Work</b>	<b>132</b>
<b>8</b>	<b>Future Work</b>	<b>133</b>
<b>9</b>	<b>Conclusions</b>	<b>134</b>
<b>10</b>	<b>Appendix - List of codes</b>	<b>136</b>
10.1	Mathematical conventions . . . . .	136
10.2	Image parameters . . . . .	137
10.3	Predictor parameters . . . . .	137
10.4	Encoder parameters . . . . .	138
10.5	VHDL Style Guide . . . . .	139
10.6	HDL Coding Guidelines . . . . .	140
10.7	SoC Package Pinout . . . . .	142
10.8	Extended Utilization Reports . . . . .	143
10.9	Signed/Unsigned signals handling . . . . .	144
10.10	VHDL Package example . . . . .	145
10.11	Image Coordinates Control IP source code . . . . .	146
10.12	Adder IP source code . . . . .	148
10.13	Scaled Difference IP source code . . . . .	150
10.14	Shift Register IP source code . . . . .	151
10.15	Image Metadata IP source code . . . . .	152
10.16	Parallel Synchronous FIFOs IP source code . . . . .	154
10.17	Sample-Adaptive GPO2 Code IP source code . . . . .	156
10.18	Adder IP Python script source code . . . . .	158
10.19	Adder IP VUnit testbench source code . . . . .	159
10.20	Simulations Bash source code . . . . .	161
10.21	Xilinx Vivado TCL framework source code . . . . .	162
10.22	Vivado Project Bash source code . . . . .	164
<b>11</b>	<b>Declaration of Authorship</b>	<b>168</b>

# List of Figures

1.1	HYPSO logo . . . . .	17
2.1	Hyper-spectral image cube [9, p.19] . . . . .	19
2.2	BSQ input order pseudo-code . . . . .	19
2.3	BI input order pseudo-code . . . . .	20
2.4	Illustration with all input sample orders [9, p.19] . . . . .	20
2.5	CCSDS-123 block diagram [10] . . . . .	21
2.6	Typical prediction neighbourhood [10] . . . . .	27
2.7	Central and neighbour samples representation . . . . .	28
2.8	Local sum options graphical representation . . . . .	30
2.9	Compressed image structure . . . . .	37
2.10	Weight Init. pseudo-code . . . . .	43
2.11	Weight Exp. Off. pseudo-code . . . . .	43
2.12	BI input order pseudo-code with error limit values . . . . .	48
2.13	Structural differences between Issue 1 and Issue 2 [29, p.3] . . . . .	55
2.14	Non-synthesizable (left) and Synthesizable (right) writing operation . . . . .	60
2.15	Non-synthesizable (left) and Synthesizable (right) For-loop . . . . .	60
3.1	CCSDS-123 Top Entity IP timing diagram . . . . .	62
3.2	Xilinx UltraScale MPSoC architecture . . . . .	64
3.3	Block diagram example . . . . .	67
3.4	Top entity IP block diagram . . . . .	68
3.5	Image coordinates IP block diagram . . . . .	71
3.6	Predictor IP block diagram . . . . .	72
3.7	Adder IP block diagram . . . . .	73
3.8	Quantizer IP block diagram . . . . .	74
3.9	Fidelity Control IP block diagram . . . . .	75
3.10	Mapper IP block diagram . . . . .	77
3.11	Sample Representative IP block diagram . . . . .	78
3.12	Double-Resolution Sample Representative IP block diagram . . . . .	79
3.13	Prediction IP block diagram . . . . .	79
3.14	Samples Store IP block diagram . . . . .	81
3.15	Shift register structure . . . . .	82
3.16	Local Sum IP block diagram . . . . .	83
3.17	Local Differences IP block diagram . . . . .	84

3.18	Local Differences Vector IP block diagram . . . . .	85
3.19	Weights Vector IP block diagram . . . . .	86
3.20	Predicted Central Local Difference IP block diagram . . . . .	88
3.21	Predictor-Encoder Interconnect IP block diagram . . . . .	89
3.22	Parallel Synchronous FIFOs IP block diagram . . . . .	90
3.23	Encoder Top IP block diagram . . . . .	91
3.24	Encoder Header IP block diagram . . . . .	92
3.25	Encoder Body IP block diagram . . . . .	96
3.26	Sample-Adaptive Entropy Coder IP block diagram . . . . .	97
3.27	Hybrid Entropy Coder IP block diagram . . . . .	98
3.28	Sample-Adaptive GPO2 Code IP block diagram . . . . .	98
3.29	Hybrid Entropy Coder IP block diagram . . . . .	100
3.30	Hybrid Low-Entropy Coder IP block diagram . . . . .	103
3.31	Hybrid Compressed Image Tail IP block diagram . . . . .	105
3.32	Packer IP block diagram . . . . .	106
3.33	Packing data timing diagram [17, p.9] . . . . .	107
4.1	GUI of testbenches bash file . . . . .	111
4.2	VUnit testbench report . . . . .	111
4.3	ModelSim Simulator Engine GUI . . . . .	112
4.4	GUI of Xilinx Vivado bash file . . . . .	113
5.1	Power consumption reports for <i>Synthesis</i> (left) and <i>Implementation</i> (right)	116
5.2	Reduced Utilization report for <i>Synthesis</i> . . . . .	117
5.3	Reduced Utilization report for <i>Implementation</i> . . . . .	117
5.4	Timing report for Synthesis . . . . .	118
5.5	Timing report for Implementation . . . . .	118
5.6	Image Coordinates Control IP waveform . . . . .	119
5.7	Adder IP waveform . . . . .	120
5.8	Quantizer IP waveform . . . . .	120
5.9	Mapper IP waveform . . . . .	121
5.10	Sample Representative IP waveform . . . . .	121
5.11	Prediction IP waveform . . . . .	122
5.12	Predictor Top IP waveform . . . . .	123
5.13	Encoder Header IP waveform . . . . .	123
5.14	Sample-Adaptive Entropy Coder IP waveform . . . . .	124
5.15	Hybrid Entropy Coder IP waveform . . . . .	124
5.16	Packer IP waveform . . . . .	125
5.17	Encoder Top IP waveform . . . . .	126
5.18	Top Entity IP waveform . . . . .	127
5.19	Gantt chart . . . . .	129
10.1	SoC XCZU9EG-FFVB1156 package pinout . . . . .	142
10.2	Extended Utilization report after Synthesis . . . . .	143

10.3	Extended Utilization report after Implementation . . . . .	143
------	--	-----

## List of Tables

2.1	Encoder Header top structure . . . . .	38
2.2	Image Metadata top . . . . .	38
2.3	Image Metadata - Essential part . . . . .	39
2.4	Supplementary Information Tables purposes . . . . .	40
2.5	Supplementary Information Table Structure . . . . .	41
2.6	Predictor Metadata top . . . . .	42
2.7	Predictor Metadata - Primary part (1/2) . . . . .	42
2.8	Predictor Metadata - Primary part (2/2) . . . . .	43
2.9	Predictor Metadata - Weight tables part . . . . .	43
2.10	Predictor Metadata - Quantization part . . . . .	44
2.11	Quantization part - Error Limit Update Period sub-part . . . . .	44
2.12	Quantization part - Absolute Error Limit sub-part . . . . .	44
2.13	Quantization part - Relative Error Limit sub-part . . . . .	45
2.14	Predictor Metadata - Sample Representative part . . . . .	46
2.15	Sample-Adaptive Entropy Coder Metadata Structure . . . . .	47
2.16	Hybrid Entropy Coder Metadata Structure . . . . .	47
2.17	Low-Entropy Code Input Symbol Limit and Threshold . . . . .	53
2.18	Constraints to turn Issue 2 into Issue 1 [10] . . . . .	56
3.1	Delay values to generate all neighbour samples . . . . .	82
10.1	Image parameters [10, p.98] . . . . .	137
10.2	Predictor parameters (1/2) [10, p.98-100] . . . . .	137
10.3	Predictor parameters (2/2) [10, p.98-100] . . . . .	138
10.4	Encoder parameters [10, p.100-101] . . . . .	138

# List of Equations

2.1	Image coordinates range $N_X, N_Y, N_Z$ parameters . . . . .	18
2.2	Image dynamic range D parameter . . . . .	18
2.3	Signed samples range . . . . .	18
2.4	Unsigned samples range . . . . .	18
2.5	Image sample identifier $s_z(t)$ value . . . . .	18
2.6	Image sample index t value . . . . .	18
2.7	Prediction residual $\Delta_z(t)$ computation . . . . .	22
2.8	Signed quantizer index $q_z(t)$ computation . . . . .	22
2.9	Maximum error $m_z(t)$ for lossless compression computation . . . . .	23
2.10	Maximum error $m_z(t)$ for absolute error limit compression computation . . . . .	23
2.11	Absolute error limit $a_z$ parameter . . . . .	23
2.12	Maximum error $m_z(t)$ for relative error limit compression computation . . . . .	23
2.13	Relative error limit $r_z$ parameter . . . . .	23
2.14	Maximum error $m_z(t)$ for absolute and relative error limit compression computation . . . . .	23
2.15	Absolute and Relative error limit constant values . . . . .	23
2.16	Mapped quantizer index $\delta_z(t)$ computation . . . . .	24
2.17	Scaled difference between $\hat{s}_z(t)$ parameter . . . . .	25
2.18	Sample representative $s_z''(t)$ computation . . . . .	25
2.19	Clipped version of the quantizer bin center $s_z'(t)$ computation . . . . .	26
2.20	Double-resolution sample representative $\tilde{s}_z(t)$ computation . . . . .	26
2.21	Sample representative resolution $\Theta_z$ parameter . . . . .	26
2.22	Sample representative damping $\varphi_z$ parameter . . . . .	26
2.23	Sample representative offset $\psi_z$ parameter . . . . .	26
2.24	Wide neighbour-oriented local sum computation . . . . .	29
2.25	Narrow neighbour-oriented local sum computation . . . . .	29
2.26	Wide column-oriented local sum computation . . . . .	29
2.27	Narrow column-oriented local sum computation . . . . .	29
2.28	Central local difference $d_z(t)$ computation . . . . .	30
2.29	North directional local differences $d_z^N(t)$ computation . . . . .	31
2.30	West directional local differences $d_z^W(t)$ computation . . . . .	31
2.31	North-West directional local differences $d_z^{NW}(t)$ computation . . . . .	31
2.32	Preceding spectral bands $P_z^*$ parameter . . . . .	31
2.33	Local difference vector $U_z(t)$ under Reduced prediction mode computation . . . . .	31
2.34	Local difference vector $U_z(t)$ under Full prediction mode computation . . . . .	32

2.35	Weight resolution $\Omega$ parameter . . . . .	32
2.36	Minimum and maximum weight values $\omega_{min}$ and $\omega_{max}$ parameters . . . . .	32
2.37	Default weight $\omega_z$ initialization under Reduced prediction mode computation . . . . .	32
2.38	Default weight $\omega_z$ initialization under Full prediction mode computation . . . . .	33
2.39	Custom weight $\omega_z$ initialization computation . . . . .	33
2.40	Weight initialization resolution Q parameter . . . . .	33
2.41	Central weight $\omega_z$ computation . . . . .	33
2.42	North directional weight $\omega_z^N(t)$ computation . . . . .	33
2.43	West directional weight $\omega_z^W(t)$ computation . . . . .	33
2.44	North-West directional weight $\omega_z^{NW}(t)$ computation . . . . .	33
2.45	Intra-band exponent offset $\varsigma_z^{(i)}$ parameter . . . . .	33
2.46	Inter-band weight exponent offset $\varsigma_z^*$ parameter . . . . .	33
2.47	Weight update scaling exponent $p(t)$ computation . . . . .	34
2.48	Weight update scaling exponent initial $v_{min}$ and final $v_{max}$ parameters . . . . .	34
2.49	Weight update scaling exponent change interval $t_{inc}$ parameter . . . . .	34
2.50	Double-resolution prediction error $e_z(t)$ computation . . . . .	34
2.51	Weights vector $W_z(t)$ in Reduced Prediction mode computation . . . . .	34
2.52	Weights vector $W_z(t)$ in Full Prediction mode computation . . . . .	35
2.53	Predicted central local difference $\hat{d}_z(t)$ computation . . . . .	35
2.54	High-resolution predicted sample $\check{s}_z(t)$ computation . . . . .	35
2.55	Register size R parameter . . . . .	35
2.56	Double-resolution predicted sample $\tilde{s}_z(t)$ computation . . . . .	36
2.57	Predicted sample $\hat{s}_z(t)$ computation . . . . .	36
2.58	Output word size B parameter . . . . .	37
2.59	Supl. Info. Table unsigned value . . . . .	40
2.60	Supl. Info. Table signed value . . . . .	40
2.61	Supl. Info. Table float 1 value . . . . .	40
2.62	Supl. Info. Table float 2 value . . . . .	40
2.63	Supl. Info. Table sign bit b value . . . . .	40
2.64	Supl. Info. Table exponent $\alpha$ value . . . . .	40
2.65	Supl. Info. Table significand j value . . . . .	40
2.66	Supl. Info. Table table bit depth $D_I$ parameter . . . . .	40
2.67	Supl. Info. Table significand bit depth $D_F$ parameter . . . . .	40
2.68	Supl. Info. Table exponent bit depth $D_E$ parameter . . . . .	40
2.69	Supl. Info. Table exponent bias $\beta$ parameter . . . . .	40
2.70	Error Limit values update ratio . . . . .	48
2.71	Accumulator $\Sigma_z(t)$ initial value . . . . .	49
2.72	Counter $\Gamma(t)$ initial value . . . . .	49
2.73	Initial count exponent $\gamma_0$ parameter . . . . .	49
2.74	Accumulator initialization table $k'_z$ parameter . . . . .	49
2.75	Accumulator $\Sigma_z(t)$ update value . . . . .	49
2.76	Counter $\Gamma(t)$ update value . . . . .	50



2.77	Rescaling counter size $\gamma^*$ parameter . . . . .	50
2.78	Unary length limit $U_{max}$ parameter . . . . .	50
2.79	Sample-Adaptive: Variable-length code $k_z(t)$ computation . . . . .	50
2.80	Sample-Adaptive: Variable-length code $k_z(t)$ parameter . . . . .	50
2.81	Counter $\Gamma(t)$ initial value . . . . .	51
2.82	High-resolution accumulator $\tilde{\Sigma}_z(t)$ initial value . . . . .	51
2.83	High-resolution accumulator $\tilde{\Sigma}_z(t)$ update value . . . . .	51
2.84	Counter $\Gamma(t)$ update value . . . . .	52
2.85	Hybrid High-Low Entropy selection computation . . . . .	52
2.86	Hybrid: Variable-length code $k_z(t)$ computation . . . . .	52
2.87	Hybrid: Variable-length code $k_z(t)$ parameter . . . . .	52
2.88	Hybrid: Code index $i$ computation . . . . .	53
2.89	Hybrid: Input symbol $\iota_z(t)$ computation . . . . .	54
5.1	Clock cycles for <i>Predictor</i> block to fully compute an incoming image . . .	127
5.2	Clock cycles for <i>Encoder</i> block to fully compute an incoming image . . .	127
5.3	Total clock cycles to fully compress an incoming image . . . . .	127
10.1	Round down function convention . . . . .	136
10.2	Round up function convention . . . . .	136
10.3	Modulus function convention . . . . .	136
10.4	Modulus*R function convention . . . . .	136
10.5	Clipping function convention . . . . .	136
10.6	Sign function convention . . . . .	136
10.7	Sign-plus function convention . . . . .	136

# Acronyms

<b>ARM</b>	Advanced RISC Machine
<b>ASIC</b>	Application-Specific Integrated Circuit
<b>AXI</b>	Advanced eXtensible Interface
<b>BSQ</b>	Band-Sequential
<b>BI</b>	Band-Interleaved
<b>BIL</b>	Band-Interleaved-by-Line
<b>BIP</b>	Band-Interleaved-by-Pixel
<b>CCSDS</b>	Consultative Committee for Space Data Systems
<b>CDC</b>	Clock Domain Crossing
<b>DUT</b>	Device Under Test
<b>ECTS</b>	European Credit Transfer System
<b>FF</b>	Flip-Flop
<b>FID</b>	Frame Interleaved by Diagonal
<b>FIFO</b>	First In First Out
<b>FPGA</b>	Field-Programmable Gate Array
<b>GPO2</b>	Golomb-Power-Of-2
<b>GUI</b>	Graphical User Interface
<b>HDL</b>	Hardware Description Language
<b>HLS</b>	High Level Synthesis
<b>HW</b>	Hardware
<b>HYPSONO</b>	Hyper-Spectral SmallSat for Ocean Observation
<b>I2C</b>	Inter-Integrated Circuit
<b>IDE</b>	Integrated Development Environment

<b>ILA</b>	Integrated Logic Analyzer
<b>IP</b>	Intellectual Property
<b>JTAG</b>	Joint Test Action Group
<b>LSb</b>	Least Significant Bit
<b>LSB</b>	Least Significant Byte
<b>LUT</b>	LookUp Table
<b>MSb</b>	Most Significant Bit
<b>MSB</b>	Most Significant Byte
<b>NTNU</b>	Norwegian University of Science and Technology
<b>OOO</b>	Out-Of-Context
<b>PL</b>	Programmable Logic
<b>PS</b>	Processing System
<b>RAM</b>	Random Access Memory
<b>RISC</b>	Reduced Instruction Set Computing
<b>RTL</b>	Register-Transfer Level
<b>SoC</b>	System On Chip
<b>SW</b>	Software
<b>TCL</b>	Tool Command Language
<b>UART</b>	Universal Asynchronous Receiver-Transmitter
<b>uC</b>	Micro-Controller
<b>VHDL</b>	VHSIC-HDL
<b>VHSIC</b>	Very High-Speed Integrated Circuit
<b>VIO</b>	Virtual Input/Output
<b>XPM</b>	Xilinx Parameterized Macro

# 1 Introduction

The space industry, slowed down since the 70's, is again accelerating thanks to the incursion of the private industry. In the recent years, companies such as SpaceX, determined to discover more from the outer space, our own planet and to step in other planets, have made the technology for the space sector to improve by leaps and bounds.

The key component for this purpose are the artificial satellites, placed in outer space to observe the surroundings by using hyper-spectral imaging. This kind of imaging allow researches to collect and process information from the electromagnetic spectrum of an image, with the objective of finding objects or identifying materials.

One example is the HYPSON mission [7], aiming to observe oceanographic phenomena by using a small satellite with a hyper-spectral camera on-board. Figure 1.1 is its logo.

This small satellite is the *SmallSat* project [8], developed at the Norwegian University of Science and Technology (NTNU): A miniaturized satellite of cubic shape, with 'low-cost' and relatively fast to develop and launch.

Many different and demanding processing tasks are executed on-board in this satellite, and due to the limited transmission speed from the antenna to the ground station (for the radio link), one critical point in the system is the image compression, meant to improve the usage of the limited data throughput and to transfer images at a reduced time [8].

From this specific issue arises the Consultative Committee for Space Data Systems (CCSDS), a group that introduces the CCSDS-123 standard, an efficient prediction-based algorithm for the compression of hyper-spectral images, characterized by its low complexity [11][10].

Two revisions of this standard have been published so far by this group: *CCSDS-123.0-B-1* (Issue 1), published in 2015, designed for the lossless compression of hyper-spectral images, and *CCSDS-123.0-B-2* (Issue 2), published in 2019, an improvement of the previous revision that offers higher compression rates by performing near-lossless compression of hyper-spectral images.

Furthermore, the quite noticeable increase on the dimensions of hyper-spectral images along the years to offer a better quality has turned out in the necessity of accelerating such algorithm by hardware, offering a way faster performance and real-time capabilities.

The chosen hardware platform to run this algorithm is the FPGA, a configurable hardware that has become the standard choice in small-satellite missions because of its reconfigurable nature and the possibility to execute complex tasks in parallel [13].

The present thesis, a direct continuation from *Specialization Project* [24], introduces a complete implementation of CCSDS-123 Issues 1 & 2 algorithms in the hardware description language VHDL, already synthesizable and prepared to run on a FPGA.

As of today, an implementation of this algorithm as complete as the present one has not been released, so this report seeks to be a reference point to any developer going into this topic. Thinking even further in the future, the source code has been designed with the reusability and modularity principles as its core, so that it can be improved or modified with a minimum effort, if needed (in case new bugs are discovered, or more revisions are released in the future, for example).

Moreover, two more things are released together with the source code: A TCL framework to automate the creation of a *Xilinx Vivado* project and the bitstream generation, and an architecture of testbenches, done with the VUnit framework, to validate all IPs.

It is highly important to highlight that the document *CCSDS-123.0-B-2* does not explain/justify the mathematics behind the CCSDS-123 algorithm effectiveness (*Green Book*), but only describes how to implement it (*Blue Book*). This is due to the fact that CCSDS group has not released such information yet.

Hereafter the document presents the following structure: *Background* explains the CCSDS-123 Issue 1 & 2 algorithms together with the differences with each other, input image data characteristics, VUnit framework and some HDL considerations. *Design* completely describes the implementation of all CCSDS-123 Issue 1 & 2 IPs and sub-IPs. *Validation Plan* describes the scope, tools and test-cases to validate the code. *Results* shows up the validation results and performance reports. *Discussion, Related Work* and *Future Work* define the current status of the work, related projects to this one and what actions to take next, respectively. Final chapters are *Conclusions* and *Appendix*.

Because of its size, all developed source code is submitted in parallel to this report, and here only specific parts from it is included in the *Appendix* section with the aim to support some explanations.



Figure 1.1: HYPSO logo

## 2 Background

Before continue reading any further below, check section 10.1 to see all mathematical conventions used here.

### 2.1 Input image

The incoming data are hyper-spectral images [19], being each one of them a three-dimensional array of data samples  $s_{z,y,x}$ , where references  $x$ ,  $y$  and  $z$  are the coordinates. The indexes  $x$  and  $y$  define the *spatial dimensions* (sample and frame), and the index  $z$  defines the *spectral band* [11][10].

These coordinates are integer values, and equation 2.1 shows their value ranges:

$$0 \leq x \leq N_X - 1 \quad 0 \leq y \leq N_Y - 1 \quad 0 \leq z \leq N_Z - 1 \quad (2.1)$$

Where  $N_x$ ,  $N_y$  and  $N_z$  values are defined between 1 and  $2^{16}$ .

All data sample have the same *dynamic range*, which is a user-specified parameter with the number of bits, limited by equation 2.2:

$$2 \leq D \leq 32 \quad (2.2)$$

And this range already denotes the minimum, middle and maximum possible values. The incoming samples can be either *signed* or *unsigned* values, and equations 2.3 and 2.4 show their limit values, respectively:

$$s_{\min} = -2^{D-1} \quad s_{\max} = 2^{D-1} - 1 \quad s_{\text{mid}} = 0 \quad (2.3)$$

$$s_{\min} = 0 \quad s_{\max} = 2^D - 1 \quad s_{\text{mid}} = 2^{D-1} \quad (2.4)$$

To make equations easier to understand, data samples and associated quantities can be identified either by the complete reference ( $x$ ,  $y$  and  $z$  indexes), or by the pair of indexes  $t$ ,  $z$ . Equation 2.5 is an example of it:

$$s_z(t) \equiv s_{z,y,x} \quad (2.5)$$

The new value  $t$  is the index of a sample inside the same *spectral band*, arranged in raster-scan order and starting with  $t = 0$ . Its conversion to the original coordinates is shown in equation 2.6:

$$t = y \cdot N_X + x \quad (2.6)$$

### 2.1.1 Input samples order

Figure 2.1 shows a hyper-spectral image cube, along with its coordinates reference:

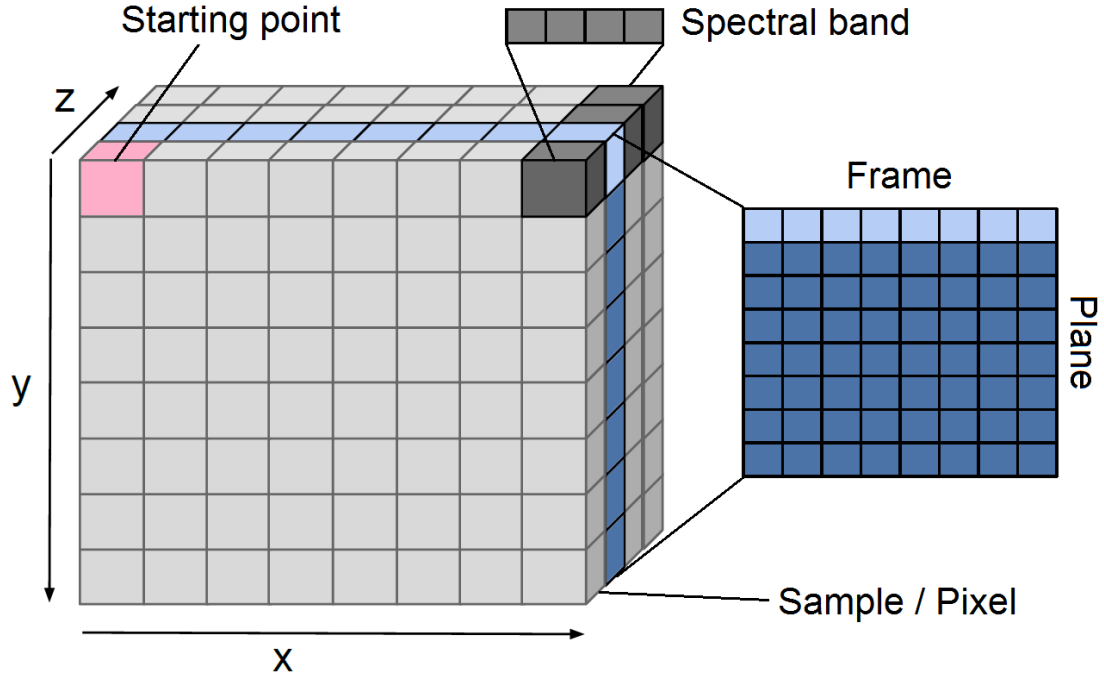


Figure 2.1: Hyper-spectral image cube [9, p.19]

When introducing an image into the system, there are two different and standard order types to enter its samples, both of them on a serial way, sample by sample: *Band-Sequential (BSQ)* and *Band-Interleaved (BI)*.

Under *BSQ*, from the starting point, samples are introduced pixel by pixel from left to right (x coordinate) until completing one frame and moving down (y coordinate), and then again for the next frame below until completing one spectral band and moving forward (z coordinate), and then again for the next spectral band in front until the cube is fully sent out.

Figure 2.2 shows the pseudo-code associated with this order:

```

for z = 0 to  $N_Z - 1$ 
  for y = 0 to  $N_Y - 1$ 
    for x = 0 to  $N_X - 1$ 
      input sample data
  
```

Figure 2.2: BSQ input order pseudo-code

Under *BI*, the samples sequence is configurable to some extent, controlled by the user-specified parameter *sub-frame interleaving depth M*, whose range is  $1 \leq M \leq N_Z$ .

Here, from the starting point, samples are introduced pixel by pixel forwards (z coordinate) until reaching  $M$  number of spectral bands and moving to the right (x coordinate), and then again for the same number of spectral bands until completing one frame and moving down (y coordinate), and then again for the next frame below until the cube is fully sent out.

Figure 2.3 shows the pseudo-code associated with this order:

```

for  $y = 0$  to  $N_Y - 1$ 
  for  $i = 0$  to  $\lceil N_Z / M \rceil - 1$ 
    for  $x = 0$  to  $N_X - 1$ 
      for  $z = iM$  to  $\min\{(i+1)M - 1, N_Z - 1\}$ 
        input sample data

```

Figure 2.3: BI input order pseudo-code

As one can see in this pseudo-code, playing with the user-specified parameter *sub-frame interleaving depth*  $M$ , it is possible from sending samples from all spectral bands before moving to the right ( $M = 1$ , special case called *Band-Interleaved-by-Pixel (BIP)*) to sending all samples from one spectral band before moving to the next one ( $M = N_Z$ , special case called *Band-Interleaved-by-Line (BIL)*).

Figure 2.4 is an illustration of the 3 sample orderings for hyper-spectral images:

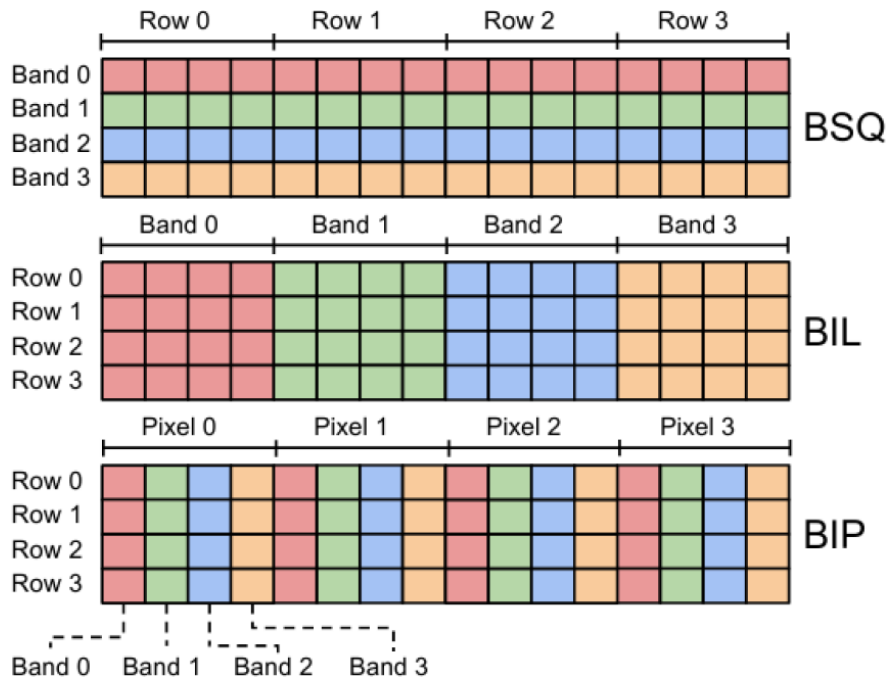


Figure 2.4: Illustration with all input sample orders [9, p.19]



## 2.2 CCSDS-123 Issue 2 algorithm

The CCSDS-123 Issue 2 algorithm is basically composed of two blocks: the *Predictor* block and the *Encoder* block. The first one takes over of predicting the new samples (based on the nearby ones), and the second one takes over the compression and codification of the image [11][10].

The two of them are placed in series, and the *Predictor* IP is the first one, in other words, this one receives the original image sample values  $s_z(t)$ .

Figure 2.5 shows the very top structure of this algorithm:

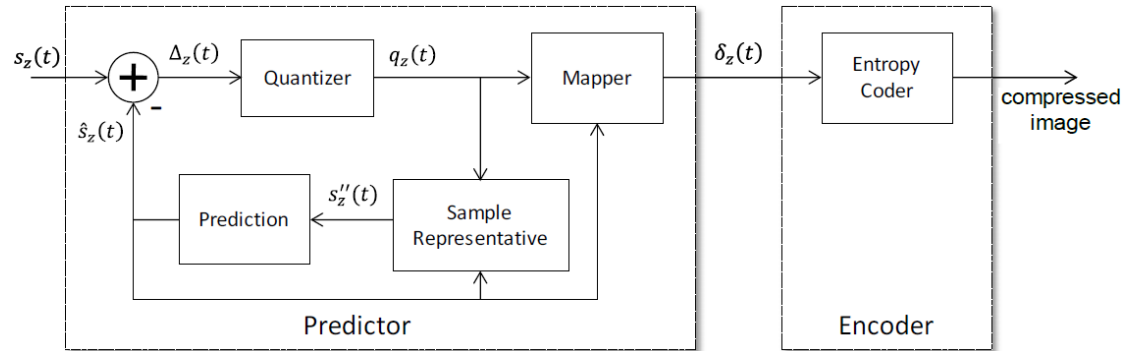


Figure 2.5: CCSDS-123 block diagram [10]

All information below about the *Predictor* and *Encoder* blocks as well as their sub-modules has been mostly extracted from [11][10].

## 2.3 Predictor block

The *Predictor* block uses an adaptive linear prediction method to predict the value of each image sample based on the nearby sample values in a small three dimensional neighborhood. This operation is computed sequentially in a single pass.

This block is composed of 5 major components or sub-blocks. The major block from the left on Figure 2.5 shows the block diagram of the *Predictor* block, together with all its interconnections.

Looking at this block from left to right and top to down: the *Adder* (top-left box, section 2.3.1) computes the *prediction residual*  $\Delta_z(t)$ , which is the difference between the *predicted sample*  $\hat{s}_z(t)$  and *original sample*  $s_z(t)$  values, and then it is quantized using a uniform *Quantizer* (top-center box, section 2.3.2). The quantizer step size can be controlled via an *absolute error limit* (samples are reconstructed with a user specified error bound), *relative error limit* (samples predicted to be smaller are reconstructed with lower error), *absolute and relative error limits* together, or totally *lossless* (obtained by setting the *absolute error limit* to zero).

To offer the near-lossless compression capability, an adaptively updated weighted *Prediction* algorithm (left-down box, section 2.3.5) is implemented in a close-loop, generating the *predicted sample*  $\hat{s}_z(t)$ .

Unlike *Issue 1* [11, p.18], the *Predictor* block cannot use the exact *original sample*  $s_z(t)$  values here because they are not available to the decompressor at the time of reconstruction, when compression is not lossless. Instead, *Prediction* calculations are computed using *sample representative*  $s_z''(t)$  values in place of *original sample*  $s_z(t)$  values, using a *Sample Representative* algorithm (right-down box, section 2.3.4), also placed in the the close-loop branch. This is required to let decompressor duplicate the prediction calculation later.

Finally, the *quantized prediction residual*  $q_z(t)$  values go through a *Mapper* (top-right box, section 2.3.3) to be mapped into *unsigned mapped quantizer index*  $\delta_z(t)$  values, which makes up the output of the *Predictor* block.

### 2.3.1 Adder

The module *Adder* is in charge of computing the very first element in the chain: the *prediction residual*  $\Delta_z(t)$ .

As the equation 2.7 shows, the *prediction residual*  $\Delta_z(t)$  is computed as the difference between the *predicted sample*  $\hat{s}_z(t)$  and *original sample*  $s_z(t)$  values:

$$\Delta_z(t) = s_z(t) - \hat{s}_z(t) \quad (2.7)$$

### 2.3.2 Quantizer

The module *Quantizer* gets the *prediction residual*  $\Delta_z(t)$  and the *predicted sample*  $\hat{s}_z(t)$  values, from modules *Adder* and *Prediction* respectively, and it produces the *maximum error value*  $m_z(t)$  and the *signed quantizer index*  $q_z(t)$  values, to use them later into modules *Mapper* and *Sample Representative*.

First, the *maximum error value*  $m_z(t)$  is computed by the sub-module *Fidelity Control* by means of the *predicted sample*  $\hat{s}_z(t)$  value, as explained in section 2.3.2.1.

The *prediction residual*  $\Delta_z(t)$  is quantized using a uniform quantizer with step size of  $2 * m_z(t) + 1$  (so controlled via the *maximum error value*  $m_z(t)$ ), generating the *signed quantizer index*  $q_z(t)$  as equation 2.8 states:

$$q_z(t) = \begin{cases} \Delta_z(0), & t = 0 \\ \text{sgn}(\Delta_z(t)) \left\lfloor \frac{|\Delta_z(t)| + m_z(t)}{2m_z(t) + 1} \right\rfloor, & t > 0 \end{cases} \quad (2.8)$$

### 2.3.2.1 Fidelity Control

The module *Fidelity Control* calculates the allowed *maximum error*  $m_z(t)$  value, based on the user settings, with the incoming *predicted sample*  $\hat{s}_z(t)$ .

For its computation, one option is to define a lossless compression, and that means fixing this error to 0, as equation 2.9 shows:

$$m_z(t) = 0 \quad (2.9)$$

A second option is to define an *absolute error limit* (so samples can be reconstructed with a user-specified error bound). This threshold is defined in equation 2.10 and constrained in equation 2.11:

$$m_z(t) = a_z \quad (2.10)$$

$$0 \leq a_z \leq 2^{D_A} - 1 \quad 1 \leq D_A \leq \min\{D - 1, 16\} \quad (2.11)$$

Another option is to define a *relative error limit* (so samples with small magnitude can be reconstructed with a preconfigured low error). This threshold is defined in equation 2.12 and constrained in equation 2.13:

$$m_z(t) = \left\lfloor \frac{r_z |\hat{s}_z(t)|}{2^D} \right\rfloor \quad (2.12)$$

$$0 \leq r_z \leq 2^{D_R} - 1 \quad 1 \leq D_R \leq \min\{D - 1, 16\} \quad (2.13)$$

Last option is to define both *absolute and relative error limits* at a time. In such case, previous constraints from equations 2.11 and 2.13 apply here again, but the equation 2.14 is the one that defines the new way to compute the *maximum error*  $m_z(t)$  value:

$$m_z(t) = \min \left( a_z, \left\lfloor \frac{r_z |\hat{s}_z(t)|}{2^D} \right\rfloor \right) \quad (2.14)$$

For any *absolute or relative error limits* are used, they can be either *band-dependent*, in which case the user must specify a value per spectral band  $z$ , or *band-independent*, in which case  $a_z = A^*$  and  $r_z = R^*$  for all spectral bands  $z$ .

User-specified parameters *absolute error limit constant*  $A^*$  and *relative error limit constant*  $R^*$  are constrained as equation 2.15 details:

$$0 \leq A^* \leq 2^{D_A} - 1 \quad 0 \leq R^* \leq 2^{D_R} - 1 \quad (2.15)$$

### 2.3.2.1.1 Periodic Error Limit Updating

The previous *absolute and relative error limit* values configuration can be either fixed for a complete image, or the user might choose to use *Periodic Error Limit Updating* option, in which case the error limit values would be periodically updated.

- This feature can only be enabled with BIP and BIL input orders, but never with BSQ input order.

With such option enabled, the user needs to provide new *Error limit values* every  $2^U$  frames, being  $U$  the user-specified parameter *error limit update period exponent*, and with range  $0 \leq U \leq 9$ .

Even though the *Error limit values* might change over time, the other quantizer fidelity settings, such as *absolute and/or relative error limits* (see equations 2.11 and 2.13), *band-dependent* or *band-independent* (see equation 2.15) or *error limit bit depth*, must always be fixed for the entire image.

### 2.3.3 Mapper

The module *Mapper* receives the *predicted sample*  $\hat{s}_z(t)$ , *maximum error*  $m_z(t)$  and *quantizer index*  $q_z(t)$  values, from modules *Prediction* and *Quantizer* respectively, and it computes the *mapped quantizer index*  $\delta_z(t)$  value: the output of *Predictor* block itself.

First, the *scaled difference between  $\hat{s}_z(t)$  and nearest endpoint  $s_{min}$ ,  $s_{max}$ ,  $\theta_z(t)$* , is computed by the sub-module *Scaled Difference* by means of the *predicted sample*  $\hat{s}_z(t)$  and *maximum error*  $m_z(t)$  values, as explained in section 2.3.3.1.

Then, the *scaled difference*  $\theta_z(t)$  value is used along with the *quantizer residual*  $q_z(t)$  to compute the *mapped quantizer index*  $\delta_z(t)$ , as equation 2.16 states:

$$\delta_z(t) = \begin{cases} |q_z(t)| + \theta_z(t), & |q_z(t)| > \theta_z(t) \\ 2|q_z(t)|, & 0 \leq (-1)^{\hat{s}_z(t)} q_z(t) \leq \theta_z(t) \\ 2|q_z(t)| - 1, & \text{otherwise} \end{cases} \quad (2.16)$$

This mapping process can be inverted, so that the decompressor can reconstruct the *quantizer index*  $q_z(t)$  with no error later on, is desired.

#### 2.3.3.1 Scaled Difference

The module *Scaled Difference* computes the *scaled difference between  $\hat{s}_z(t)$  and nearest endpoint  $s_{min}$ ,  $s_{max}$ ,  $\theta_z(t)$* , value, calculated based in the *predicted sample*  $\hat{s}_z(t)$  and the *maximum error*  $m_z(t)$  values.

Equation 2.17 defines the calculation of this value:

$$\theta_z(t) = \begin{cases} \min \{ \hat{s}_z(0) - s_{\min}, s_{\max} - \hat{s}_z(0) \} & t = 0 \\ \min \left\{ \left\lfloor \frac{\hat{s}_z(t) - s_{\min} + m_z(t)}{2m_z(t) + 1} \right\rfloor, \left\lfloor \frac{s_{\max} - \hat{s}_z(t) + m_z(t)}{2m_z(t) + 1} \right\rfloor \right\} & t > 0 \end{cases} \quad (2.17)$$

### 2.3.4 Sample Representative

As already said on section 2.3, the *Predictor* block in Issue 2 cannot work with the exact values of the *original sample*  $s_z(t)$  signal, because they are not available to the decompressor for reconstruction when the compression is not lossless. Instead, *Prediction* calculations must be performed using a *sample representative*  $s_z''(t)$  in place of the *original sample*  $s_z(t)$  values.

The module *Sample Representative* receives the *original sample*  $s_z(t)$  (from outside), *predicted sample*  $\hat{s}_z(t)$ , *high-resolution predicted sample*  $\check{s}_z(t)$  (from module *Prediction*), *maximum error*  $m_z(t)$  and *quantizer index*  $q_z(t)$  (from module *Mapper*) values, producing the *sample representative*  $s_z''(t)$  and *clipped quantizer bin center*  $s_z'(t)$  values, both of them sent to the module *Prediction*.

To begin with, the *clipped version of the quantizer bin center*  $s_z'(t)$  is computed by means of the *predicted sample*  $\hat{s}_z(t)$ , *quantizer index*  $q_z(t)$  and *maximum error*  $m_z(t)$  values, as explained on section 2.3.4.1 below.

The next step is to compute the *double-resolution sample representative*  $\tilde{s}_z(t)$  by means of the *clipped version of the quantizer bin center*  $s_z'(t)$ , *quantizer index*  $q_z(t)$ , *maximum error*  $m_z(t)$  and *high-resolution predicted sample*  $\check{s}_z(t)$  values, as explained on section 2.3.4.2 below.

Finally, the *double-resolution sample representative*  $\tilde{s}_z(t)$  is used along with the *original sample*  $s_z(t)$  to produce the *sample representative*  $s_z''(t)$  value (which has the same resolution as the *original sample*  $s_z(t)$ ), as equation 2.18 states:

$$s_z''(t) = \begin{cases} s_z(0), & t = 0 \\ \left\lfloor \frac{\tilde{s}_z''(t) + 1}{2} \right\rfloor, & t > 0 \end{cases} \quad (2.18)$$

### 2.3.4.1 Clipped Quantizer Bin Center

This module computes the *clipped version of the quantizer bin center*  $s'_z(t)$  value, based on the *predicted sample*  $\hat{s}_z(t)$ , *quantizer index*  $q_z(t)$  and *maximum error*  $m_z(t)$  values.

Equation 2.19 states how to calculate the *clipped quantizer bin center*  $s'_z(t)$ :

$$s'_z(t) = \text{clip}\left(\hat{s}_z(t) + q_z(t)(2m_z(t) + 1), \{s_{\min}, s_{\max}\}\right) \quad (2.19)$$

In point of fact, the reconstruction of the *original sample*  $s_z(t)$  by the decompressor using the *clipped quantizer bin center*  $s'_z(t)$  value ensures that the error during such process will be at most  $m_z(t)$ . That is to say, if decompression is chosen to be lossless (equation 2.9), it means that these two values will always be the same ( $s'_z(t) = s_z(t)$ ).

Refer to section 2.5 to see more information about this particular configuration.

### 2.3.4.2 Double-Resolution Sample Representative

This module calculates the *double-resolution sample representative*  $\tilde{s}_z(t)$  value, based on the *clipped version of the quantizer bin center*  $s'_z(t)$ , *quantizer index*  $q_z(t)$ , *maximum error*  $m_z(t)$  and *high-resolution predicted sample*  $\check{s}_z(t)$  values, as equation 2.20 shows:

$$\tilde{s}_z''(t) = \left\lfloor \frac{4(2^\Theta - \phi_z) \cdot (s'_z(t) \cdot 2^\Omega - \text{sgn}(q_z(t)) \cdot m_z(t) \cdot \psi_z \cdot 2^{\Omega-\Theta}) + \phi_z \cdot \check{s}_z(t) - \phi_z \cdot 2^{\Omega+1}}{2^{\Omega+\Theta+1}} \right\rfloor \quad (2.20)$$

Moreover, equations 2.21, 2.22 and 2.23 show the user-specified parameters involved in previous equation 2.20: *sample representative resolution* ( $\Theta_z$ ), *damping* ( $\varphi_z$ ) and *offset* ( $\psi_z$ ), and their constraints:

$$0 \leq \Theta \leq 4 \quad (2.21) \quad 0 \leq \phi_z \leq 2^\Theta - 1 \quad (2.22) \quad 0 \leq \psi_z \leq 2^\Theta - 1 \quad (2.23)$$

If lossless compression is used,  $\psi_z$  is directly fixed to 0. Besides, as one can appreciate from equations 2.19 and 2.20 show, setting  $\varphi_z = \psi_z = 0$  causes that  $s''_z(t) = s'_z(t)$ .

## 2.3.5 Prediction

The module *Prediction* receives the *clipped quantizer bin center*  $s'_z(t)$  and *sample representative*  $s''_z(t)$  values from the *Sample Representative*, and the *original sample*  $s_z(t)$  value from the external sensor, and it computes the *high-resolution predicted sample*  $\check{s}_z(t)$  value for the *Sample Representative*, and the *predicted sample*  $\hat{s}_z(t)$  value, used by all the other modules.

The user can choose to execute this algorithm in either *reduced* or *full prediction mode*, except when the image has a width of one ( $N_X = 1$ ), when only the *reduced prediction mode* shall be used.

Under both prediction modes, module *Prediction* makes use of the *central local differences*, only from the parameter *preceding spectral bands*  $P_z^*$  (see equation 2.32). But for *full prediction mode*, it additionally uses the *three directional (neighbour) local differences* as well (see sections 2.3.5.2, 2.3.5.3 and 2.3.5.7).

Figure 2.6 illustrates the typical neighbourhood in the module *Prediction*:

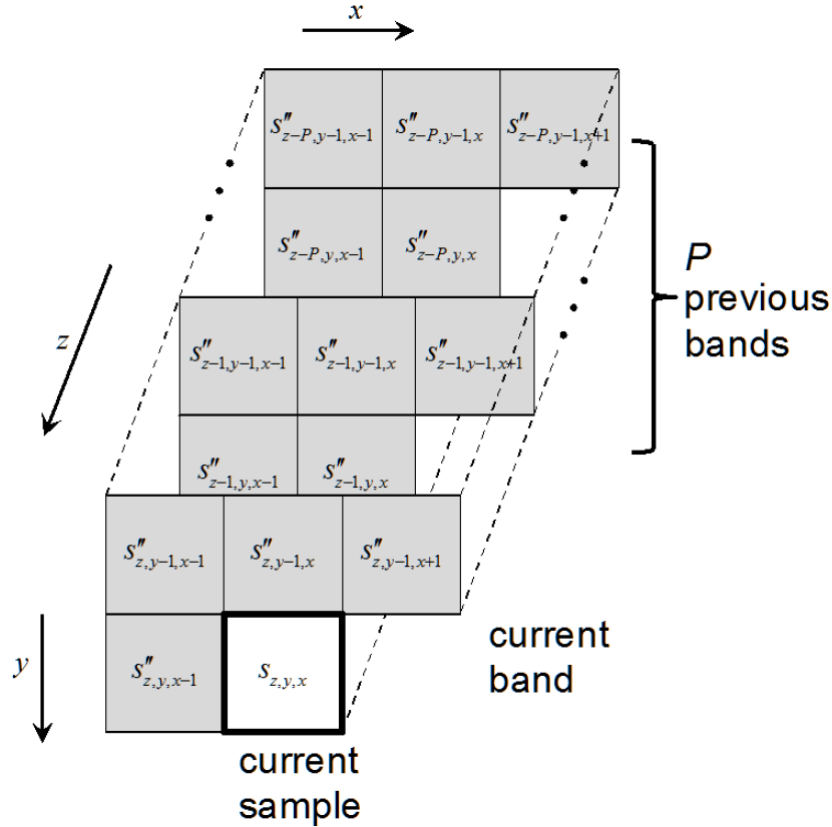


Figure 2.6: Typical prediction neighbourhood [10]

Prediction can be performed causally in a single pass through the image, broken down into the following sorted out steps:

1. Inside each *spectral band*, a *local sum*  $\sigma_z(t)$  of neighboring *sample representative*  $s_z''(t)$  values are computed (see section 2.3.5.1), and these two signals are then taken to compute the *central plus directional*, if configured so, *local difference*  $d_z(t)$  values (see section 2.3.5.2).
2. The *clipped quantizer bin center*  $s_z'(t)$  and *double-resolution predicted sample*  $\tilde{s}_z(t)$  values are used to produce the *double-resolution prediction error*  $e_z(t)$  value (see section 2.3.5.6), and together with the *weight update scaling exponent*  $p(t)$  (see section 2.3.5.5), so the *central (plus directional, if configured so) weight*  $\omega_z$  values are computed (see section 2.3.5.4).

3. The *central and directional local difference*  $d_z(t)$  and *weight*  $\omega_z$  values are put together into vectors  $U_z(t)$  and  $W_z(t)$  (see sections 2.3.5.3 and 2.3.5.7, respectively) to compute the *predicted central local difference*  $\hat{d}_z(t)$  value, as described on section 2.3.5.8.
4. With the resulting *predicted central local difference*  $\hat{d}_z(t)$  value and the previous *local sum*  $\sigma_z(t)$  value, the *high-resolution predicted sample*  $\check{s}_z(t)$  value is computed (see section 2.3.5.9), being one of the outputs of the module *Prediction*.
5. The *high-resolution predicted sample*  $\check{s}_z(t)$  value is, by means of the previous *original sample*  $s_z(t)$  value, upgraded to the *double-resolution predicted sample*  $\tilde{s}_z(t)$  value (see section 2.3.5.10), the second output of the module *Prediction*.
6. Finally, the previous *double-resolution predicted sample*  $\tilde{s}_z(t)$  value is upgraded once more to compute the final *predicted sample*  $\hat{s}_z(t)$  value (see section 2.3.5.11), being this one the third and last output of the module *Prediction*.

It is quite interesting to highlight steps 2 & 5, where the *double-resolution predicted sample*  $\tilde{s}_z(t)$  value turns out to be a backwards-dependency in order to update the *central and directional weight*  $\omega_z$  values. That means the module *Prediction*, in the same way as the *Predictor* block, has a close-loop design.

### 2.3.5.1 Local Sum

The *local sum*  $\sigma_z(t)$  is a weighted sum of some previous *sample representative*  $s_z''(t)$  values in *spectral band*  $z$  that are neighbours to the current one.

More specifically, such required neighbour samples are: North-West (NW), North (N), North-East (NE) and West (W). Figure 2.7 shows a graphical representation of the current (central) and these neighbour samples:

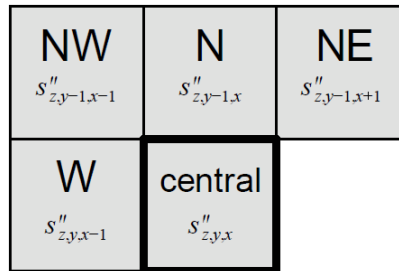


Figure 2.7: Central and neighbour samples representation

There are 4 different possibilities to compute this sum, each one of them with specific neighbour dependencies, and it is configurable by the user:



- *Wide neighbour-oriented local sum* option (equation 2.24):

$$\sigma_{z,y,x} = \begin{cases} s''_{z,y,x-1} + s''_{z,y-1,x-1} + s''_{z,y-1,x} + s''_{z,y-1,x+1}, & y > 0, 0 < x < N_X - 1 \\ 4s''_{z,y,x-1}, & y = 0, x > 0 \\ 2(s''_{z,y-1,x} + s''_{z,y-1,x+1}), & y > 0, x = 0 \\ s''_{z,y,x-1} + s''_{z,y-1,x-1} + 2s''_{z,y-1,x}, & y > 0, x = N_X - 1 \end{cases} \quad (2.24)$$

- *Narrow neighbour-oriented local sum* option (equation 2.25):

$$\sigma_{z,y,x} = \begin{cases} s''_{z,y-1,x-1} + 2s''_{z,y-1,x} + s''_{z,y-1,x+1}, & y > 0, 0 < x < N_X - 1 \\ 4s''_{z-1,y,x-1}, & y = 0, x > 0, z > 0 \\ 2(s''_{z,y-1,x} + s''_{z,y-1,x+1}), & y > 0, x = 0 \\ 2(s''_{z,y-1,x-1} + s''_{z,y-1,x}), & y > 0, x = N_X - 1 \\ 4s_{mid}, & y = 0, x > 0, z = 0 \end{cases} \quad (2.25)$$

- *Wide column-oriented local sum* option (equation 2.26):

$$\sigma_{z,y,x} = \begin{cases} 4s''_{z,y-1,x}, & y > 0 \\ 4s''_{z,y,x-1}, & y = 0, x > 0 \end{cases} \quad (2.26)$$

- *Narrow column-oriented local sum* option (equation 2.27):

$$\sigma_{z,y,x} = \begin{cases} 4s''_{z,y-1,x}, & y > 0 \\ 4s''_{z-1,y,x-1}, & y = 0, x > 0, z > 0 \\ 4s_{mid}, & y = 0, x > 0, z = 0 \end{cases} \quad (2.27)$$

For a better understanding of the equations listed above, Figure 2.8 shows the *neighbour sample representative*  $s''_z(t)$  values dependencies, depending on the selected sum option (only applicable when both X and Y coordinates are bigger than 0):

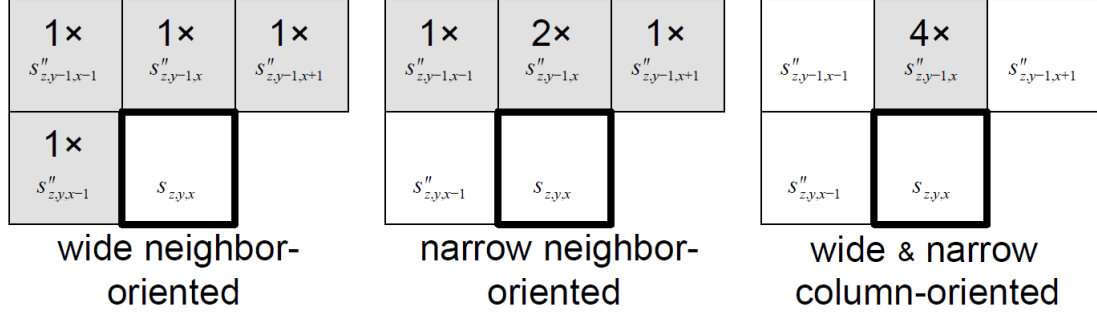


Figure 2.8: Local sum options graphical representation

The use of *reduced prediction mode* in combination with *column-oriented local sums* offers smaller compressed image data volumes for raw (uncalibrated) input images from push-broom imagers that exhibit significant along-track streaking artifacts [10, p.27]. On the other side, the use of *full prediction mode* in combination with *neighbor-oriented local sums* offers smaller compressed image data volumes for whisk-broom imagers, frame imagers, and calibrated imagery [10, p.27].

Nevertheless, there are a couple of constraints when selecting one of these 4 possible operations:

- If the image has a width of 1 ( $N_X = 1$ ), which also involves using *reduced prediction mode*, *column-oriented* (either wide or narrow) *local sums* should be used.
- Under *full prediction mode*, *neighbour-oriented* (either wide or narrow) *local sums* should be used.

*Narrow local sums* are defined to eliminate the dependency on *sample representative*  $s''_{z,y,x-1}$  value when calculating the *local sum*  $\sigma_z(t)$  value, which may facilitate pipelining in a hardware implementation [10, p.26].

### 2.3.5.2 Local Differences

The *local sum*  $\sigma_z(t)$  together with the *current and neighbour sample representative*  $s''_z(t)$  values are used to produce the *central and directional local difference*  $d_z(t)$  values.

Hereafter, only directions North (N), West (W) and North-West (NW) are required.

The *central local difference*  $d_z(t)$  value is the difference between *local sum*  $\sigma_z(t)$  and four times the *current sample representative*  $s''_z(t)$  values, as equation 2.28 shows <sup>1</sup>.

$$d_{z,y,x} = 4s''_{z,y,x} - \sigma_{z,y,x} \quad (2.28)$$

<sup>1</sup>Only applicable when  $t > 0$ .

The *three directional local differences*  $d_z^N(t)$ ,  $d_z^W(t)$  and  $d_z^{NW}(t)$  values are the differences between *local sum*  $\sigma_z(t)$  and four times the corresponding *neighbour sample representative*  $s_z''(t)$  values, just as equations 2.29, 2.30 and 2.31 show <sup>1</sup>:

$$d_{z,y,x}^N = \begin{cases} 4s_{z,y-1,x}'' - \sigma_{z,y,x}, & y > 0 \\ 0, & y = 0 \end{cases} \quad (2.29)$$

$$d_{z,y,x}^W = \begin{cases} 4s_{z,y,x-1}'' - \sigma_{z,y,x}, & x > 0, y > 0 \\ 4s_{z,y-1,x}'' - \sigma_{z,y,x}, & x = 0, y > 0 \\ 0, & y = 0 \end{cases} \quad (2.30)$$

$$d_{z,y,x}^{NW} = \begin{cases} 4s_{z,y-1,x-1}'' - \sigma_{z,y,x}, & x > 0, y > 0 \\ 4s_{z,y-1,x}'' - \sigma_{z,y,x}, & x = 0, y > 0 \\ 0, & y = 0 \end{cases} \quad (2.31)$$

The graphical representation from Figure 2.7 helps understand the location of all *neighbour sample representative*  $s_z''(t)$  values.

As already said on section 2.3.5, the *central local difference*  $d_z(t)$  value is always used, regardless of the system configuration, but the *directional local difference*  $d_z^N(t)$ ,  $d_z^W(t)$  and  $d_z^{NW}(t)$  values are only used under *full prediction mode*.

### 2.3.5.3 Local Differences Vector

The *local difference vector*  $U_z(t)$  is composed of the *central local difference*  $d_z(t)$  values from the *preceding spectral bands*  $P_z^*$  and *directional local differences*  $d_z^N(t)$ ,  $d_z^W(t)$  and  $d_z^{NW}(t)$  values.

The parameter  $P_z^*$  directly depends on the the user-specified parameter *number of prediction bands*  $P$  (range  $0 \leq P \leq 15$ ), as the equation 2.32 defines:

$$P_z^* = \min \{z, P\} \quad (2.32)$$

When working under *reduced prediction mode*, the *local difference vector*  $U_z(t)$  includes only the *central local difference*  $d_z(t)$  values, as equation 2.33 indicates:

$$\mathbf{U}_z(t) = \begin{bmatrix} d_{z-1}(t) \\ d_{z-2}(t) \\ \vdots \\ d_{z-P_z^*}(t) \end{bmatrix} \quad (2.33)$$

But if working under *full prediction mode*, the *local difference vector*  $U_z(t)$  will also include the *directional local differences*  $d_z^N(t)$ ,  $d_z^W(t)$  and  $d_z^{NW}(t)$  values, placed at the beginning of it, as equation 2.34 indicates:

$$\mathbf{U}_z(t) = \begin{bmatrix} d_z^N(t) \\ d_z^W(t) \\ d_z^{NW}(t) \\ d_{z-1}(t) \\ d_{z-2}(t) \\ \vdots \\ d_{z-P_z^*}(t) \end{bmatrix} \quad (2.34)$$

#### 2.3.5.4 Weight values

Each *central/directional local difference*  $d_z(t)$  value from the *local difference vector*  $U_z(t)$  is multiplied by a *weight*  $\omega_z$  value, and both operands are adaptively updated following the calculation of each *predicted sample*  $\hat{s}_z(t)$  value.

All *weight*  $\omega_z$  values are *signed*, whose resolution is within the range  $\Omega + 3$  bits, being this user-specified parameter constrained as equation 2.35 shows:

$$4 \leq \Omega \leq 19 \quad (2.35)$$

Of course, an increase in the number of bits for representing the *weight*  $\omega_z$  values also provides an increased resolution in the *Prediction* calculation.

This resolution means that each *weight*  $\omega_z$  value has a minimum and maximum possible value  $\omega_{min}$  and  $\omega_{max}$  values, respectively, shown on equation 2.36:

$$\omega_{min} = -2^{\Omega+2} \quad \omega_{max} = 2^{\Omega+2} - 1 \quad (2.36)$$

In order to initialize the *central and directional weight*  $\omega_z$  values, the user can decide by either *default* or *custom weight initialization*, and such option shall be applied on each *spectral band*  $z$ .

For a *default weight initialization*, the *central weight*  $\omega_z$  values from the *preceding spectral bands*  $P_z^*$  are calculated according to equation 2.37:

$$\omega_z^{(1)}(1) = \frac{7}{8} 2^\Omega, \quad \omega_z^{(i)}(1) = \left\lfloor \frac{1}{8} \omega_z^{(i-1)}(1) \right\rfloor, \quad i = 2, 3, \dots, P_z^* \quad (2.37)$$

Moreover, in case *full prediction mode* is selected, the *directional weight*  $\omega_z^N(t)$ ,  $\omega_z^W(t)$  and  $\omega_z^{NW}(t)$  values are set to 0, as equation 2.38 states:

$$\omega_z^N(1) = \omega_z^W(1) = \omega_z^{NW}(1) = 0 \quad (2.38)$$

On the other hand, in case *custom weight initialization* is chosen, and regardless of the selected prediction mode, both the *central weight*  $\omega_z$  values from the *preceding spectral bands*  $P_z^*$  and *directional weight*  $\omega_z^N(t)$ ,  $\omega_z^W(t)$  and  $\omega_z^{NW}(t)$  values are calculated according to the equation 2.39:

$$\mathbf{W}_z(1) = 2^{\Omega+3-Q} \Lambda_z + \lceil 2^{\Omega+2-Q} - 1 \rceil \mathbf{1} \quad (2.39)$$

The symbol  $\mathbf{1}$  denotes a vector of all ‘ones’, and equation 2.40 indicates the constraints of the user-specified parameter *weight initialization resolution*  $Q$  (in bits):

$$3 \leq Q \leq \Omega + 3 \quad (2.40)$$

After (any of) these initializations, the *weight*  $\omega_z$  values must be updated every clock cycle. Thus, they are computed following equation 2.41 for *central weight*  $\omega_z$  values, and equations for 2.42, 2.43 and 2.44 for *directional weight*  $\omega_z^N(t)$ ,  $\omega_z^W(t)$  and  $\omega_z^{NW}(t)$  values:

$$\omega_z^{(i)}(t+1) = \text{clip} \left( \omega_z^{(i)}(t) + \left\lfloor \frac{1}{2} \left( \text{sgn}^+ [e_z(t)] \cdot 2^{-(\rho(t)+\zeta_z^{(i)})} \cdot d_{z-i}(t) + 1 \right) \right\rfloor, \{\omega_{\min}, \omega_{\max}\} \right) \quad (2.41)$$

$$\omega_z^N(t+1) = \text{clip} \left( \omega_z^N(t) + \left\lfloor \frac{1}{2} \left( \text{sgn}^+ [e_z(t)] \cdot 2^{-(\rho(t)+\zeta_z^*)} \cdot d_z^N(t) + 1 \right) \right\rfloor, \{\omega_{\min}, \omega_{\max}\} \right) \quad (2.42)$$

$$\omega_z^W(t+1) = \text{clip} \left( \omega_z^W(t) + \left\lfloor \frac{1}{2} \left( \text{sgn}^+ [e_z(t)] \cdot 2^{-(\rho(t)+\zeta_z^*)} \cdot d_z^W(t) + 1 \right) \right\rfloor, \{\omega_{\min}, \omega_{\max}\} \right) \quad (2.43)$$

$$\omega_z^{NW}(t+1) = \text{clip} \left( \omega_z^{NW}(t) + \left\lfloor \frac{1}{2} \left( \text{sgn}^+ [e_z(t)] \cdot 2^{-(\rho(t)+\zeta_z^*)} \cdot d_z^{NW}(t) + 1 \right) \right\rfloor, \{\omega_{\min}, \omega_{\max}\} \right) \quad (2.44)$$

Equations 2.45 and 2.46 show the constraints of two user-specified parameters used above: the *Intra-band exponent offset*  $\zeta_z^{(i)}$  and *Inter-band weight exponent offset*  $\zeta_z^*$ .

Be aware that the second parameter is just a value, but the first one is an array, with a different value per *spectral band*  $z$ :

$$-6 \leq \zeta_z^{(i)} \leq 5 \quad (2.45)$$

$$-6 \leq \zeta_z^* \leq 5 \quad (2.46)$$

Last but not least, see sections 2.3.5.5 and 2.3.5.6 for the computation of *weight update scaling exponent*  $p(t)$  and *double-resolution prediction error*  $e_z(t)$  values, respectively, also used in the previous equations.

### 2.3.5.5 Weight Update Scaling Exponent

The *weight update scaling exponent*  $p(t)$  value is computed simply by using the *image coordinates* ( $t$  component) and some user-specified parameters defined below, as equation 2.47 shows:

$$\rho(t) = \text{clip} \left( v_{\min} + \left\lfloor \frac{t - N_X}{t_{\text{inc}}} \right\rfloor, \{v_{\min}, v_{\max}\} \right) + D - \Omega \quad (2.47)$$

A small value of  $p(t)$  produces a big weight increment, giving a faster adaptation to source statistics as a result, but worse steady-state compression performance [10, p.37].

The user-specified parameters *weight update scaling exponent initial*  $v_{\min}$ , *weight update scaling exponent final*  $v_{\max}$  and *weight update scaling exponent change interval*  $t_{\text{inc}}$  are constrained according to equations 2.48 and 2.49:

$$-6 \leq v_{\min} \leq v_{\max} \leq 9 \quad (2.48) \quad 2^4 \leq t_{\text{inc}} \leq 2^{11} \text{ (power of 2)} \quad (2.49)$$

These parameters define the ratio at which the *weight*  $\omega_z$  values adapt to the image data statistics.

### 2.3.5.6 Double-Resolution Prediction Error

The *double-resolution prediction error*  $e_z(t)$  value is computed as the difference between two times the *clipped quantizer bin center*  $s'_z(t)$  and *double-resolution predicted sample*  $\tilde{s}_z(t)$  values, as equation 2.50 shows:

$$e_z(t) = 2s'_z(t) - \tilde{s}_z(t) \quad (2.50)$$

### 2.3.5.7 Weights Vector

The *weights vector*  $W_z(t)$  is made of the *central weight*  $\omega_z$  values from the *preceding spectral bands*  $P_z^*$  and *directional weight*  $\omega_z^N(t)$ ,  $\omega_z^W(t)$  and  $\omega_z^{NW}(t)$  values.

When working under *reduced prediction mode*, the *weights vector*  $W_z(t)$  includes only the *central weight*  $\omega_z(t)$  values, as equation 2.51 indicates:

$$\mathbf{W}_z(t) = \begin{bmatrix} \omega_z^{(1)}(t) \\ \omega_z^{(2)}(t) \\ \vdots \\ \omega_z^{(P_z^*)}(t) \end{bmatrix} \quad (2.51)$$

But if working under *full prediction mode*, the *weights vector*  $W_z(t)$  will also include the *directional weights*  $\omega_z^N(t)$ ,  $\omega_z^W(t)$  and  $\omega_z^{NW}(t)$  values, located at the beginning of it, as equation 2.52 indicates:

$$\mathbf{W}_z(t) = \begin{bmatrix} \omega_z^N(t) \\ \omega_z^W(t) \\ \omega_z^{NW}(t) \\ \omega_z^{(1)}(t) \\ \omega_z^{(2)}(t) \\ \vdots \\ \omega_z^{(P_z^*)}(t) \end{bmatrix} \quad (2.52)$$

### 2.3.5.8 Predicted Central Local Difference

The *predicted central local difference*  $\hat{d}_z(t)$  value is the inner product between the *local difference vector*  $U_z(t)$  and *weight vector*  $W_z(t)$ . In other words, each *central/directional local difference sample*  $\hat{d}_z(t)$  component is multiplied by the corresponding *central/directional weight*  $\omega_z$  component, and then all of them are summed.

The result is just one single value per vectors multiplication, as equation 2.53 shows:

$$\hat{d}_z(t) = \mathbf{W}_z^T(t) \mathbf{U}_z(t) \quad (2.53)$$

This equation is always applied, except when being under *reduced prediction mode* and working with the first *spectral band* ( $z = 0$ ). In such case:  $\hat{d}_z(t) = 0$ .

### 2.3.5.9 High-Resolution Predicted Sample

The *high-resolution predicted sample*  $\check{s}_z(t)$  value is computed with the *predicted central local difference*  $\hat{d}_z(t)$  and *local sum*  $\sigma_z(t)$  values, as equation 2.54 states:

$$\check{s}_z(t) = \text{clip} \left( \text{mod}_R^* \left[ \hat{d}_z(t) + 2^\Omega (\sigma_z(t) - 4s_{\text{mid}}) \right] + 2^{\Omega+2} s_{\text{mid}} + 2^{\Omega+1}, \left\{ 2^{\Omega+2} s_{\text{min}}, 2^{\Omega+2} s_{\text{max}} + 2^{\Omega+1} \right\} \right) \quad (2.54)$$

where the user-specified parameter *register size*  $R$  is constrained as equation 2.55 shows:

$$\max\{32, D + \Omega + 2\} \leq R \leq 64 \quad (2.55)$$

Note that increasing the *register size*  $R$  reduces the chance of an overflow during the *high-resolution predicted sample*  $\check{s}_z(t)$  value computation [10, p.33].

### 2.3.5.10 Double-Resolution Predicted Sample

The *double-resolution predicted sample*  $\tilde{s}_z(t)$  value is computed by means of the *original sample*  $s_z(t)$  from the previous *spectral band*  $z$  and *high-resolution predicted sample*  $\check{s}_z(t)$  values, as equation 2.56 states:

$$\tilde{s}_z(t) = \begin{cases} \left\lfloor \frac{\check{s}_z(t)}{2^{\Omega+1}} \right\rfloor, & t > 0 \\ 2s_{z-1}(t), & t = 0, P > 0, z > 0 \\ 2s_{\text{mid}}, & t = 0 \text{ and } (P = 0 \text{ or } z = 0) \end{cases} \quad (2.56)$$

### 2.3.5.11 Predicted Sample

Finally, the *double-resolution predicted sample*  $\tilde{s}_z(t)$  value is upgraded with a simple operation to produce the *predicted sample*  $\hat{s}_z(t)$ , just as equation 2.57 indicates:

$$\hat{s}_z(t) = \left\lfloor \frac{\tilde{s}_z(t)}{2} \right\rfloor \quad (2.57)$$



## 2.4 Encoder block

The *Encoder* block losslessly encodes an input image, by compressing the incoming *mapped quantizer index*  $\delta_z(t)$  values from the *Predictor* block, along with some input image and compression parameters on the top of it as a header.

The major block on the right side from Figure 2.5 is the block diagram of the *Encoder* block, and it is composed of 2 major components or sub-blocks: the *Encoder Header* and the *Encoder Body*.

A compressed image consists of a variable-length header followed by a variable-length body. Its simple structure is defined on Figure 2.9:



Figure 2.9: Compressed image structure

The variable-length *Encoder Header* (see section 2.4.1) encodes the image and compression parameters. These entropy coder parameters are adaptively adjusted along the way to adapt to changes in the statistics of the *mapped quantizer index*  $\delta_z(t)$  values.

After that, the variable-length *Encoder Body* (see section 2.4.2) losslessly encodes *mapped quantizer index*  $\delta_z(t)$  values. Additionally, if the *Periodic Error Limit Updating* option is enabled (see section 2.3.2.1.1), then *Error limit values* are periodically encoded as part of the body too. In such case, the samples input order BIP/BIL (because this option is not allowed under BSQ) is updated as shown in Figure 2.12.

For the encoding process, the user can choose to perform it using the *Sample-Adaptive Entropy Coder* (see section 2.4.2.1), *Hybrid Entropy Coder* (see section 2.4.2.2) or *Block-Adaptive Entropy Coder* (which could not be added for the present work).

The *Sample-Adaptive Entropy Coder* and *Block-Adaptive Entropy Coder* approaches are inherited from Issue 1 [11], and so, they are effective for lossless compression.

Nevertheless, the feature of near-lossless compression, introduced in Issue 2 [10], tends to yield *mapped quantizer index*  $\delta_z(t)$  values having a lower-entropy distribution. For such configuration, the new *Hybrid Entropy Coder* approach tends to provide more effective encoding for lower-entropy distributions [10, p.39].

With the data successfully encoded, the output signal width is the user-specified parameter *output word size*  $B$ , measured in bytes, constrained according to equation 2.58:

$$1 \leq B \leq 8 \tag{2.58}$$

If necessary, 0-padding bits should be included in the body to ensure that the size of the output compressed image is the proper one.

### 2.4.1 Encoder Header

The header of a compressed image always have the same structure (but different configuration), and it is sorted out as follows:

1. Image Metadata (see section 2.4.1.1).
2. Predictor Metadata (see section 2.4.1.2).
3. Entropy Coder Metadata (see section 2.4.1.3).

Part	Status	Size
Image Metadata	Mandatory	Variable
Predictor Metadata	Mandatory	Variable
Entropy Coder Metadata	Mandatory	Variable

Table 2.1: Encoder Header top structure

Starting with Table 2.1, all the header metadata must be structured with the same order as the tables below list their different fields.

The size of each header part depends on the selected compression options, and it is not necessarily a multiple of the *output word size*  $B$ , but in case fill bits are required at any place, 0-padding bits should be used.

#### 2.4.1.1 Image Metadata

Table 2.2 shows the top structure of the *Image Metadata*:

Subpart	Status	Size (Bytes)
Essential	Mandatory	12
Supplementary Information Tables	Optional	Variable

Table 2.2: Image Metadata top

It can be seen that this metadata is divided into 2 groups:

1. Essential: General information about the incoming image, with a fixed size of 12-bytes. Its structure is depicted on Table 2.3.
2. Supplementary Information Tables: Optional table(s) that provide(s) auxiliary information of the image, and with a variable size, depending on the user configuration (see section 2.4.1.1.1).

Field	Width (bits)	Description
User-Defined Data	8	The user may assign the value of this field arbitrarily, for example, to indicate the value of a user-defined index of the image within a sequence of images.
X Size	16	The value $N_X$ encoded mod $2^{16}$ as a 16-bit unsigned binary integer.
Y Size	16	The value $N_Y$ encoded mod $2^{16}$ as a 16-bit unsigned binary integer.
Z Size	16	The value $N_Z$ encoded mod $2^{16}$ as a 16-bit unsigned binary integer.
Sample Type	1	'0': image sample values are unsigned integers. '1': image sample values are signed integers.
Reserved	1	This field shall have value '0'.
Large Dynamic Range Flag	1	'0': dynamic range satisfies $D \leq 16$ . '1': dynamic range satisfies $D > 16$ .
Dynamic Range	4	The value $D$ mod $2^4$ as a 4-bit unsigned binary integer.
Sample Encoding Order	1	'0': samples are encoded in band-interleaved order. '1': samples are encoded in BSQ order.
Sub-Frame Interleaving Depth	16	When band-interleaved encoding order is used, this field shall contain the value $M$ encoded mod $2^{16}$ as a 16-bit unsigned binary integer. When BSQ encoding order is used, this field shall be all 'zeros'.
Reserved	2	This field shall have value '00'.
Output Word Size	3	The value $B$ encoded mod $2^3$ as a 3-bit unsigned binary integer.
Entropy Coder Type	2	'00': sample-adaptive entropy coder is used. '01': hybrid entropy coder is used. '10': block-adaptive entropy coder is used.
Reserved	1	This field shall have value '0'.
Quantizer Fidelity Control Method	2	'00': lossless. '01': absolute error limit only. '10': relative error limit only. '11': both absolute and relative error limits.
Reserved	2	This field shall contain all 'zeros'.
Supplementary Information Table Count	4	The value $\tau$ , encoded as a 4-bit unsigned integer.

Table 2.3: Image Metadata - Essential part

### 2.4.1.1.1 Supplementary Information Tables

The CCSDS-123 algorithm allows to define up to 15 (user-specified parameter  $\tau$ ) *Supplementary Information Tables* in order to encode them as part of the *Image Metadata*, offering auxiliary image information to an end user.

The user must configure each table with a specific *purpose*, *structure* and *data type*.

For the table purpose, the possibilities are listed on Table 2.4:

Purpose	Interpretation
0	scale
1	offset
2	wavelength
3	full width at half maximum
4	defect indicator
5–9	reserved
10–15	user-defined

Table 2.4: Supplementary Information Tables purposes

Regarding the table structure, it can be:

- Zero-dimensional table, so just a single element.
- One-dimensional table, having  $N_Z$  elements (one for each spectral band  $z$ ).
- Two-dimensional table, having  $N_Z * N_X$  elements (one for each  $[z, x]$  component).
- Two-dimensional table, having  $N_Y * N_X$  elements (one for each  $[y, x]$  component).

Finally for the table data type, expected possibilities are:

- Unsigned value  $i$ , with the configurable range defined on equation 2.59:
- Signed value  $i$ , with the configurable range defined on equation 2.60:

$$0 \leq i \leq 2^{D_I} - 1 \quad (2.59) \quad -2^{D_I-1} \leq i \leq 2^{D_I-1} - 1 \quad (2.60)$$

- Float values composed of *sign bit*  $b$ ,  $\alpha$  and *significand*  $j$ , with a configurable range defined on equation 2.61 when  $\alpha = 0$ , or on equation 2.62 when  $\alpha > 0$ :

$$(-1)^b \cdot j \cdot 2^{1-\beta-D_F} \quad (2.61) \quad (-1)^b \left( 2^{D_F} + j \right) 2^{\alpha-\beta-D_F} \quad (2.62)$$

Equations 2.63 to 2.69 show the constraints for user-specified parameters *table bit depth*  $D_I$ , *significand bit depth*  $D_F$ , *exponent bit depth*  $D_E$ , *exponent bias*  $\beta$ , *sign bit*  $b$ , *exponent*  $\alpha$  and *significand*  $j$ :

$$0 \leq b \leq 1 \quad (2.63) \quad 0 \leq \alpha \leq 2^{D_E} - 1 \quad (2.64) \quad 0 \leq j \leq 2^{D_F} - 1 \quad (2.65)$$

$$1 \leq D_I \leq 32 \quad (2.66) \quad 2 \leq D_E \leq 8 \quad (2.67) \quad 1 \leq D_F \leq 23 \quad (2.68) \quad 0 \leq \beta \leq 2^{D_E} - 1 \quad (2.69)$$

With all *Supplementary Information Tables* already defined, Table 2.5 shows how each one of them are integrated within the *Image Metadata*:

Field	Width (bits)	Description
Table Type	2	'00': unsigned integer. '01': signed integer. '10': float.
Reserved	2	This field shall have value '00'.
Table Purpose	4	Table purpose value encoded as a 4-bit unsigned integer (see table 3-1).
Reserved	1	This field shall have value '0'.
Table Structure	2	'00': zero-dimensional. '01': one-dimensional. '10': two-dimensional-zx. '11': two-dimensional-yx.
Reserved	1	This field shall have value '0'.
Supplementary User-Defined Data	4	The user may assign the value of this field arbitrarily.
Table Data Subblock	(variable)	(See below.)

Table 2.5: Supplementary Information Table Structure

The *Table Data Subblock* depends on the user configuration, so its size is variable:

- If the table *data type* is signed or unsigned:
  1. User-specified parameter *table bit depth*  $D_I$  encoded modulo  $2^5$  (so 5 bits).
  2. All elements of each table encoded, with  $D_I$  bits each, always in increasing index order (nested loops for the two-dimensional cases).
  3. If necessary, 0-bits padding to reach the next word boundary.
- If the table *data type* is float:
  1. User-specified parameter *significand bit depth*  $D_F$  encoded modulo  $2^5$  (5 bits).
  2. User-specified parameter *exponent bit depth*  $D_E$  encoded modulo  $2^3$  (3 bits).
  3. User-specified parameter *exponent bias*  $\beta$  encoded modulo  $2^{D_E}$  ( $D_E$  bits).
  4. All elements of each table encoded, with  $1 + D_F + D_E$  bits each, always in increasing index order (nested loops for the two-dimensional cases).
  5. For every single element encoded, its associated parameters *sign bit*  $b$  (1 bit), *exponent*  $\alpha$  ( $D_E$  bits) and *significand*  $j$  ( $D_F$  bits) are encoded too.
  6. If necessary, 0-bits padding to reach the next word boundary.

### 2.4.1.2 Predictor Metadata

Table 2.6 shows the top structure of the *Predictor Metadata*:

Subpart	Status	Size (Bytes)
Primary	Mandatory	5
Weight Tables	Optional	Variable
Quantization	Conditional	Variable
Sample Representative	Conditional	Variable

Table 2.6: Predictor Metadata top

The first group of the *Predictor Metadata* is the *Primary* part, which is mandatory and it has a fixed size of 5 bytes. Tables 2.7 and 2.8 show its structure:

Field	Width (bits)	Description
Reserved	1	This field shall have value '0'.
Sample Representative Flag	1	'0': Sample Representative subpart is not included in Predictor Metadata header part; sample representatives use $\phi_z = \psi_z = 0$ for all spectral bands $z$ . '1': Sample Representative subpart is included in Predictor Metadata header part.
Number of Prediction Bands	4	The value $P$ encoded as a 4-bit unsigned binary integer.
Prediction Mode	1	'0': full prediction mode is used. '1': reduced prediction mode is used.
Weight Exponent Offset Flag	1	'0': all $\zeta_z^{(i)}$ and $\zeta_z^*$ values are zero. '1': some $\zeta_z^{(i)}$ and $\zeta_z^*$ values may be nonzero.
Local Sum Type	2	'00': wide neighbor-oriented local sums are used. '01': narrow neighbor-oriented local sums are used. '10': wide column-oriented local sums are used. '11': narrow column-oriented local sums are used.
Register Size	6	The value $R$ encoded mod $2^6$ as a 6-bit unsigned binary integer.
Weight Component Resolution	4	The value $(\Omega - 4)$ encoded as a 4-bit unsigned binary integer.
Weight Update Scaling Exponent Change Interval	4	The value $(\log_2 t_{mc} - 4)$ encoded as a 4-bit unsigned binary integer.
Weight Update Scaling Exponent Initial Parameter	4	The value $(v_{\min} + 6)$ encoded as a 4-bit unsigned binary integer.
Weight Update Scaling Exponent Final Parameter	4	The value $(v_{\max} + 6)$ encoded as a 4-bit unsigned binary integer.

Table 2.7: Predictor Metadata - Primary part (1/2)

Field	Width (bits)	Description
Weight Exponent Offset Table Flag	1	'0': Weight Exponent Offset Table is not included in Predictor Metadata. '1': Weight Exponent Offset Table is included in Weight Tables subpart of Predictor Metadata.
Weight Initialization Method	1	'0': default weight initialization is used. '1': custom weight initialization is used.
Weight Initialization Table Flag	1	'0': Weight Initialization Table is not included in Predictor Metadata. '1': Weight Initialization Table is included in Weight Tables subpart of Predictor Metadata.
Weight Initialization Resolution	5	When the default weight initialization is used, this field shall have value '00000'. Otherwise, this field shall contain the value $Q$ encoded as a 5-bit unsigned binary integer.

Table 2.8: Predictor Metadata - Primary part (2/2)

The next group is the *Weight Tables* part, which is divided into 2 sub-groups, both of them optional and with a variable size, just as Table 2.9 shows:

Block	Status	Size
Weight Initialization Table	Optional	Variable
Weight Exponent Offset Table	Optional	Variable

Table 2.9: Predictor Metadata - Weight tables part

The sub-group *Weight Initialization Table* is only instantiated when *Weight Initialization Table Flag* is asserted. If so, it consists on encoding each component from the *custom weight initialization vector*  $\Lambda_z$  (see equation 2.39) as two's component signed value, and using  $Q$ -bits for each, nesting loops in increasing index order as Figure 2.10 shows.

The sub-group *Weight Exponent Offset Table* is only instantiated when *Weight Exponent Offset Table Flag* is asserted. If so, it consists on encoding each component from the *Intra-band weight exponent offsets*  $\varsigma_z^{(i)}$  and *Inter-band weight exponent offsets*  $\varsigma_z^*$  (see equations 2.45 and 2.46) as two's component signed value, and using 4-bits for each, nesting loops in increasing index order as Figure 2.11 shows.

<pre> for z = 0 to <math>N_z - 1</math>   for j = 0 to <math>C_z - 1</math>     encode component j of <math>\Lambda_z</math> </pre>	<pre> for z = 0 to <math>N_z - 1</math>   if full prediction mode used     encode <math>\varsigma_z^*</math>     for i = 1 to <math>P_z^*</math>       encode <math>\varsigma_z^{(i)}</math> </pre>
---	---

Figure 2.10: Weight Init. pseudo-code

Figure 2.11: Weight Exp. Off. pseudo-code

Another group is the *Quantization* part, divided into 3 sub-groups. All of them are conditional, and even though the first chunk has a fixed size of 1 byte, the others have variable size. Table 2.10 shows its structure:

Block	Status	Size (Bytes)
Error Limit Update Period	Conditional	1
Absolute Error Limit	Conditional	Variable
Relative Error Limit	Conditional	Variable

Table 2.10: Predictor Metadata - Quantization part

The sub-group *Error Limit Update Period* must be integrated only if the *Periodic Error Limit Updating* option is enabled (see section 2.3.2.1.1). Its structure is depicted on Table 2.11, with a fixed size of 1 byte:

Field	Width (bits)	Description
Reserved	1	This field shall have value '0'.
Periodic Updating Flag	1	'0': periodic error limit updating is not used. '1': periodic error limit updating is used.
Reserved	2	This field shall contain all 'zeros'.
Update Period Exponent	4	When periodic error limit updating is used, this field shall contain the value $u$ encoded as a 4-bit unsigned binary integer. Otherwise, this field shall contain all 'zeros'.

Table 2.11: Quantization part - Error Limit Update Period sub-part

The sub-group *Absolute Error Limit* must be integrated only in case the *Absolute Error Limit* option is enable for compression (see section 2.3.2.1). Its structure is shown on Table 2.12:

Field	Width (bits)	Description
Reserved	1	This field shall have value '0'.
Absolute Error Limit Assignment Method	1	'0': band-independent absolute error limit assignment. '1': band-dependent absolute error limit assignment.
Reserved	2	This field shall have value '00'.
Absolute Error Limit Bit Depth	4	The value $D_A$ encoded mod $2^4$ as a 4-bit unsigned integer.
Absolute Error Limit Values Subblock (conditional)	(variable)	(See below.)

Table 2.12: Quantization part - Absolute Error Limit sub-part



The field *Absolute Error Limit Values Subblock* is included only if the *Periodic Error Limit Updating* option is disabled. Its structure is (see section 2.3.2.1):

- If *band-independent* configured, it consists of user-specified parameter *absolute error limit constant*  $A^*$  encoded with  $D_A$ -bits.
- If *band-dependent* configured, it consists of the sequence of  $a_z$  values, in order of increasing band index  $z$ , each encoded with  $D_A$ -bits.

If necessary, 0-padding bits shall be appended until reaching the next word boundary.

The sub-group *Relative Error Limit* must be integrated only in case the *Relative Error Limit* option is enable for compression (see section 2.3.2.1). Its structure is shown on Table 2.13:

Field	Width (bits)	Description
Reserved	1	This field shall have value '0'.
Relative Error Limit Assignment Method	1	'0': band-independent relative error limit assignment. '1': band-dependent relative error limit assignment.
Reserved	2	This field shall have value '00'.
Relative Error Limit Bit Depth	4	The value $D_R$ encoded mod $2^4$ as a 4-bit unsigned integer.
Relative Error Limit Values Subblock (conditional)	(variable)	(See below.)

Table 2.13: Quantization part - Relative Error Limit sub-part

The field *Relative Error Limit Values Subblock* is included only if the *Periodic Error Limit Updating* option is disabled. Its structure is (see section 2.3.2.1):

- If *band-independent* configured, it consists of user-specified parameter *relative error limit constant*  $R^*$  encoded with  $D_R$ -bits.
- If *band-dependent* configured, it consists of the sequence of  $r_z$  values, in order of increasing band index  $z$ , each encoded with  $D_R$ -bits.

If necessary, 0-padding bits shall be appended until reaching the next word boundary.

Last group is the *Sample Representative* part, with no sub-divisions on it. Table 2.14 depicts its structure:

Field	Width (bits)	Description
Reserved	5	This field shall contain all 'zeros'.
Sample Representative Resolution	3	Value of $\Theta$ encoded as a 3-bit unsigned binary integer.
Reserved	1	This field shall have value '0'.
Band-Varying Damping Flag	1	'0': all bands use the same value of $\phi_z$ . '1': the value $\phi_z$ of may vary from band to band.
Damping Table Flag	1	'0': the Damping Table subblock is not included in the Sample Representative subpart. '1': the Damping Table subblock is included in the Sample Representative subpart.
Reserved	1	This field shall have value '0'.
Fixed Damping Value	4	If the Band-Varying Damping Flag field is '0', then this field encodes the value of $\phi_z$ to use for all bands as a 4-bit unsigned integer. Otherwise, this field shall be all 'zeros'.
Reserved	1	This field shall have value '0'.
Band-Varying Offset Flag	1	'0': all bands use the same value of $\psi_z$ . '1': the value of $\psi_z$ may vary from band to band.
Offset Table Flag	1	'0': the Offset Table subblock is not included in the Sample Representative subpart. '1': the Offset Table subblock is included in the Sample Representative subpart.
Reserved	1	This field shall have value '0'.
Fixed Offset Value	4	If the Band-Varying Offset Field Flag field is '0', then this field encodes the value of $\psi_z$ to use for all bands as a 4-bit unsigned integer. Otherwise, this field shall be all 'zeros'.
Damping Table Subblock (optional)	(variable)	(See below.)
Offset Table Subblock (optional)	(variable)	(See below.)

Table 2.14: Predictor Metadata - Sample Representative part

The field *Damping Table Subblock* must be included only if the *Damping Table Flag* is asserted. In such case, it consists of the sequence of *damping*  $\varphi_z$  values, in order of increasing band index  $z$ , with  $\Theta_z$ -bits each (see section 2.3.4.2).

The field *Offset Table Subblock* must be included only if the *Offset Table Flag* is asserted. In such case, it consists of the sequence of *offset*  $\psi_z$  values, in order of increasing band index  $z$ , with  $\Theta_z$ -bits each (see section 2.3.4.2).

### 2.4.1.3 Encoder Metadata

The *Encoder Metadata* just consists on adding the configuration of the selected Entropy Coder, and that means data from Table 2.15 is appended in case the *Sample-Adaptive Entropy Coder* has been enabled, or data from Table 2.16 is appended in case the *Hybrid Entropy Coder* has been enabled <sup>2</sup>.

Field	Width (bits)	Description
Unary Length Limit	5	The value $U_{\max}$ encoded mod $2^5$ as a 5-bit unsigned binary integer.
Rescaling Counter Size	3	The value $(\gamma^* - 4)$ encoded as a 3-bit unsigned binary integer.
Initial Count Exponent	3	The value $\gamma_0$ encoded mod $2^3$ as a 3-bit unsigned binary integer.
Accumulator Initialization Constant	4	When an accumulator initialization constant $K$ is specified, this field encodes the value of $K$ as a 4-bit unsigned binary integer. Otherwise, this field shall be all 'ones'.
Accumulator Initialization Table Flag	1	'0': Accumulator Initialization Table is not included in Entropy Coder Metadata. '1': Accumulator Initialization Table is included in Entropy Coder Metadata.
Accumulator Initialization Table (Optional)	(variable)	(See below.)

Table 2.15: Sample-Adaptive Entropy Coder Metadata Structure

Field	Width (bits)	Description
Unary Length Limit	5	The value $U_{\max}$ encoded mod $2^5$ as a 5-bit unsigned binary integer.
Rescaling Counter Size	3	The value $(\gamma^* - 4)$ encoded as a 3-bit unsigned binary integer.
Initial Count Exponent	3	The value $\gamma_0$ encoded mod $2^3$ as a 3-bit unsigned binary integer.
Reserved	5	This field shall have value '00000'.

Table 2.16: Hybrid Entropy Coder Metadata Structure

Both encoder metadata options have a static number of data bits to include, except the *Accumulator Initialization Table* from the *Sample-Adaptive Entropy Coder*, which is conditional to the *Accumulator Initialization Table Flag*. If such flag is asserted, this field is added into the metadata, and it consists of the concatenated sequence of  $k_z''$  values (see equation 2.74), each one of them encoded as 4-bits unsigned.

<sup>2</sup>*Block-Adaptive Entropy Coder* metadata table not included here, because it was not implemented.

## 2.4.2 Encoder Body

As mentioned on section 2.4, the module *Encoder Body* is in charge of losslessly encoding the incoming *mapped quantizer index*  $\delta_z(t)$  values, together with the *Error limit values*, in case the *Periodic Error Limit Updating* option is enabled.

The encoding process is performed by using either the *Sample-Adaptive Entropy Coder* (see section 2.4.2.1) or *Hybrid Entropy Coder* (see section 2.4.2.2) <sup>3</sup>.

The image samples come into the module as shown in Figures 2.2 and 2.3, but in case the *Periodic Error Limit Updating* option is enabled, something that is only possible under BI input order (see section 2.3.2.1.1), Figure 2.3 is upgraded into Figure 2.12:

```

for  $y = 0$  to  $N_Y - 1$ 
  if  $y \bmod 2^u = 0$  and periodic error limit updating is used
    if absolute error limits are used
      if absolute error limits are band-independent
        input  $A^*$  to the entropy coder
      else
        input  $a_0, a_1, \dots, a_{N_Z-1}$  to the entropy coder
    if relative error limits are used
      if relative error limits are band-independent
        input  $R^*$  to the entropy coder
      else
        input  $r_0, r_1, \dots, r_{N_Z-1}$  to the entropy coder
  for  $i = 0$  to  $\lceil N_Z / M \rceil - 1$ 
    for  $x = 0$  to  $N_X - 1$ 
      for  $z = iM$  to  $\min\{(i+1)M - 1, N_Z - 1\}$ 
        input  $\delta_{z,y,x}$  to the entropy coder

```

Figure 2.12: BI input order pseudo-code with error limit values

The upgrade consists in, apart from the *mapped quantizer index*  $\delta_z(t)$  values, providing also new absolute and/or relative error limit value(s) once every  $U$  frames ( $y$  coordinate), as equation 2.70 details:

$$y \bmod 2^u = 0 \quad (2.70)$$

Regardless of the selected entropy coder, each *absolute* and *relative error limit value* is encoded as  $D_A$ -bit and  $D_R$ -bit unsigned respectively, remaining the selection statistics (see sections 2.4.2.1.1 and 2.4.2.2.1) unaffected while these are processed.

---

<sup>3</sup>*Block-Adaptive Entropy Coder* is not mentioned here, because it was not implemented.

### 2.4.2.1 Sample-Adaptive Entropy Coder

The module *Sample-Adaptive Entropy Coder* encodes each incoming *mapped quantizer index*  $\delta_z(t)$  and *Error Limit Values* using a variable-length binary codeword  $R_{k_z(t)}(j)$  from a family of codes (see section 2.4.2.1.2).

The chosen member of this family is adaptively selected based on statistics that are updated after each encoding, keeping one for every *spectral band* (see section 2.4.2.1.1).

#### 2.4.2.1.1 Sample-Adaptive Statistic

The module *Sample-Adaptive Statistics* receives both the *mapped quantizer index*  $\delta_z(t)$  and *Error Limit Values* and it computes the *accumulator*  $\Sigma_z(t)$  and *counter*  $\Gamma(t)$  values. They are adaptively updated along the way, and its ratio  $\Sigma_z(t)/\Gamma(t)$  is an estimation of the *mean mapped quantizer index*  $\delta_z(t)$  value in the current spectral band, which determines the variable-length codeword to use next.

Equations 2.71 and 2.72 define the initial values of the *accumulator*  $\Sigma_z(t)$  and *counter*  $\Gamma(t)$ , respectively:

$$\Sigma_z(1) = \left\lfloor \frac{1}{2^7} (3 \cdot 2^{k_z'+6} - 49) \Gamma(1) \right\rfloor \quad \Gamma(1) = 2^{\gamma_0} \quad (2.71) \quad (2.72)$$

The user-specified parameters *initial count exponent*  $\gamma_0$  and *accumulator initialization table*  $k_z'$  from above are constrained here in equations 2.73 and 2.74:

$$1 \leq \gamma_0 \leq 8 \quad (2.73)$$

$$k_z' = \begin{cases} k_z'', & k_z'' \leq 30 - D \\ 2k_z'' + D - 30, & k_z'' > 30 - D \end{cases} \quad 0 \leq k_z'' \leq \min(D - 2, 14) \quad (2.74)$$

Moreover, the user-specified parameter *Accumulator Initialization Constant*  $K$  is used as the initial value of  $k_z''$ , ensuring this initial value for the parameter *variable-length codeword*  $k_z(t)$ .

After initialization, and for  $t > 1$ , equations 2.75 and 2.76 define the next values of the *accumulator*  $\Sigma_z(t)$  and *counter*  $\Gamma(t)$ , rescaling based on condition:

$$\Sigma_z(t) = \begin{cases} \Sigma_z(t-1) + \delta_z(t-1), & \Gamma(t-1) < 2^{\gamma^*} - 1 \\ \left\lfloor \frac{\Sigma_z(t-1) + \delta_z(t-1) + 1}{2} \right\rfloor, & \Gamma(t-1) = 2^{\gamma^*} - 1 \end{cases} \quad (2.75)$$

$$\Gamma(t) = \begin{cases} \Gamma(t-1)+1, & \Gamma(t-1) < 2^{\gamma^*} - 1 \\ \left\lfloor \frac{\Gamma(t-1)+1}{2} \right\rfloor, & \Gamma(t-1) = 2^{\gamma^*} - 1 \end{cases} \quad (2.76)$$

Both equations from above show that the *accumulator*  $\Sigma_z(t)$  and *counter*  $\Gamma(t)$  re-scaling is controlled by the user-specified parameter *rescaling counter size*  $\gamma^*$ , constrained according to equation 2.77:

$$\max\{4, \gamma_0 + 1\} \leq \gamma^* \leq 11 \quad (2.77)$$

#### 2.4.2.1.2 Sample-Adaptive GPO2 Coder .

The module *Sample-Adaptive GPO2 Coder* generates length-limited binary codewords  $R_{k_z(t)}(j)$  by means of the *mapped quantizer index*  $\delta_z(t)$ , *Error Limit Values*, *accumulator*  $\Sigma_z(t)$  and *counter*  $\Gamma(t)$  values.

This codeword is a length-limited Golomb-Power-Of-2 (GPO2) codeword, denoted  $R_{k_z(t)}(j)$  and being  $j$  the *mapped quantizer index*  $\delta_z(t)$ . It is defined as follows:

- If  $\lfloor j/2^{k_v(t)} \rfloor < U_{max}$ , then  $R_{k_z(t)}(j)$  is  $\lfloor j/2^{k_v(t)} \rfloor$  'zeros', followed by a 'one', and finally the  $k_v(t)$  least significant bits of the  $j$  value.
- Otherwise,  $R_{k_z(t)}(j)$  consists of  $U_{max}$  'zeros', followed by the D-bits of  $j$  value.

The user-specified parameter *unary length limit*  $U_{max}$  is limited as equation 2.78 says:

$$8 \leq U_{max} \leq 32 \quad (2.78)$$

When  $t > 0$ , the output codeword is  $R_{k_z(t)}(\delta_z(t))$  as shown above, but for the first *mapped quantizer index* in each *spectral band*  $z$  ( $t = 0 \rightarrow \delta_z(0)$ ), the output is uncoded.

The parameter *variable-length code*  $k_z(t)$  is computed by using the *accumulator*  $\Sigma_z(t)$  and *counter*  $\Gamma(t)$  values within equation 2.79, and constrained according to equation 2.80:

$$\Gamma(t)2^{k_z(t)} \leq \Sigma_z(t) + \left\lfloor \frac{49}{2^7} \Gamma(t) \right\rfloor \quad (2.79)$$

$$0 \leq k_z(t) \leq D - 2 \quad (2.80)$$

In case the very last codeword in the compressed image does not fully reach the *output word*  $B$  boundary, 0-padding bits shall be appended as needed.

### 2.4.2.2 Hybrid Entropy Coder

Similar to the *Sample-Adaptive Entropy Coder*, the module *Hybrid Entropy Coder* encodes the incoming *mapped quantizer index*  $\delta_z(t)$  with a variable-length family of codes, but in this case there are more possibilities.

The codewords can be either 'high-entropy', equivalent to those used by the *Sample-Adaptive Entropy Coder* (see section 2.4.2.2.2), or 'low-entropy', additional 16 variable-to-variable length codewords (see section 2.4.2.2.3).

When a 'high-entropy' code is selected, it immediately produces an output codeword to the bitstream, but if 'low-entropy' code is selected, it waits until enough data has arrived to determine the next output codeword. This possibility of encoding multiple *mapped quantizer index*  $\delta_z(t)$  values with just a single output codeword, makes 'low-entropy' code reach lower compressed datarates compared to 'high-entropy' code [10, p.39].

Like the *Sample-Adaptive Entropy Coder* again, the *Hybrid Entropy Coder* also uses adaptive code selection statistics in order to assign each *mapped quantizer index*  $\delta_z(t)$  to either the 'high-entropy' or 'low-entropy' coding method (see section 2.4.2.2.1).

Once compression is finished, the image body ends with a image 'tail', which encodes the final state of each 'low-entropy' flush code and the final *high-resolution accumulator*  $\tilde{\Sigma}_z(t)$  value for each spectral band (see section 2.4.2.2.4).

#### 2.4.2.2.1 Hybrid Statistic

The module *Hybrid Statistics* receives both the *mapped quantizer index*  $\delta_z(t)$  and *Error Limit Values* and computes the *high-resolution accumulator*  $\tilde{\Sigma}_z(t)$  and *counter*  $\Gamma(t)$  values. They are adaptively updated along the way, and its ratio  $\tilde{\Sigma}_z(t)/\Gamma(t)$  is an estimation of the *mean mapped quantizer index*  $\delta_z(t)$  value in the current spectral band, which determines how  $\delta_z(t)$  is encoded next.

Equation 2.81 defines the initial value of the *counter*  $\Gamma(t)$ :

$$\Gamma(0) = 2^{\gamma_0} \quad (2.81)$$

Regarding the *high-resolution accumulator*  $\tilde{\Sigma}_z(t)$ , its initial value should be within the range  $0 \leq \tilde{\Sigma}_z(0) < 2^{D+\gamma_0}$ , but there is no specific initial value for it, unless there is already an estimation of  $\delta_z(t)$  (coming from preceding compressed images), in which case equation 2.82 defines a reasonable starting point:

$$\tilde{\Sigma}_z(0) = 4\Gamma(0)\hat{\delta}_z \quad (2.82)$$

After initialization, and for  $t \geq 1$ , equations 2.83 and 2.84 define the updated values of the *high-resolution accumulator*  $\tilde{\Sigma}_z(t)$  and *counter*  $\Gamma(t)$ :

$$\tilde{\Sigma}_z(t) = \begin{cases} \tilde{\Sigma}_z(t-1) + 4\delta_z(t), & \Gamma(t-1) < 2^{\gamma^*} - 1 \\ \left\lfloor \frac{\tilde{\Sigma}_z(t-1) + 4\delta_z(t) + 1}{2} \right\rfloor, & \Gamma(t-1) = 2^{\gamma^*} - 1 \end{cases} \quad (2.83)$$

$$\Gamma(t) = \begin{cases} \Gamma(t-1)+1, & \Gamma(t-1) < 2^{\gamma^*} - 1 \\ \left\lfloor \frac{\Gamma(t-1)+1}{2} \right\rfloor, & \Gamma(t-1) = 2^{\gamma^*} - 1 \end{cases} \quad (2.84)$$

When  $\Gamma(t-1) = 2^{\gamma^*} - 1$  (or code statistics are rescaled, as seen on equations 2.83 and 2.84), the least-significant bit of  $\tilde{\Sigma}_z(t-1)$  is encoded in the bitstream, right before the output codeword defined in sections 2.4.2.2.2 and 2.4.2.2.3. Such bit allows the decoder to reconstruct the sequence of *high-resolution accumulator*  $\tilde{\Sigma}_z(t)$  values [10, p.63].

The first *mapped quantizer index* in each *spectral band*  $z$  ( $t = 0 \rightarrow \delta_z(0)$ ) is always uncoded, but when  $t > 0$ , the current *high-resolution accumulator*  $\tilde{\Sigma}_z(t)$  and *counter*  $\Gamma(t)$  values are applied into the following equation 2.85:

$$\tilde{\Sigma}_z(t) \cdot 2^{14} \geq T_0 \cdot \Gamma(t) \quad (2.85)$$

If this mathematical condition holds true, then  $\delta_z(t)$  is a '*high-entropy*' *mapped quantizer index* and it is encoded according to section 2.4.2.2.2. Otherwise, or when  $D = 2$  too,  $\delta_z(t)$  is a '*low-entropy*' *mapped quantizer index* and it is encoded according to section 2.4.2.2.3.

#### 2.4.2.2.2 Hybrid High-Entropy Coder

Quite similar to Sample-Adaptive GPO2 Coder, the module *Hybrid High-Entropy Coder* generates length-limited binary codewords by means of the *mapped quantizer index*  $\delta_z(t)$ , *Error Limit Values*, *high-resolution accumulator*  $\tilde{\Sigma}_z(t)$  and *counter*  $\Gamma(t)$  values.

This codeword is a reversed length-limited Golomb-Power-Of-2 (GPO2) codeword, denoted  $R'_{k_z(t)}(j)$  and being  $j$  the *mapped quantizer index*  $\delta_z(t)$ . It is defined as follows:

- If  $\lfloor j/2^{k_v(t)} \rfloor < U_{max}$ , then  $R'_{k_z(t)}(j)$  is the  $k_v(t)$  least significant bits of the  $j$  value, followed by a 'one', and finally  $\lfloor j/2^{k_v(t)} \rfloor$  'zeros'.
- Otherwise,  $R'_{k_z(t)}(j)$  consists of the D-bits of  $j$  value, followed by  $U_{max}$  'zeros'.

The parameter *variable-length code*  $k_z(t)$  is computed by using the *high-resolution accumulator*  $\tilde{\Sigma}_z(t)$  and *counter*  $\Gamma(t)$  values within equation 2.86, and constrained according to equation 2.87:

$$\Gamma(t)2^{k_z(t)+2} \leq \tilde{\Sigma}_z(t) + \left\lfloor \frac{49}{2^5} \Gamma(t) \right\rfloor \quad (2.86)$$

$$0 \leq k_z(t) \leq \max\{D-2, 2\} \quad (2.87)$$

In case the very last codeword in the compressed image does not fully reach the *output word*  $B$  boundary, 0-padding bits shall be appended as needed.



### 2.4.2.2.3 Hybrid Low-Entropy Coder

The module *Hybrid Low-Entropy Coder* generates one of 16 variable-to-variable length codewords by means of the *mapped quantizer index*  $\delta_z(t)$ , *Error Limit Values*, *high-resolution accumulator*  $\tilde{\Sigma}_z(t)$  and *counter*  $\Gamma(t)$  values.

A single low-entropy codeword can encode multiple *mapped quantizer index*  $\delta_z(t)$  values, allowing lower compressed data rates compared to the high-entropy codes.

Each low-entropy codeword consists of:

- A *threshold value*  $T_i$  and *input symbol limit*  $L_i$ , given on Table 2.17.
- 16 tables with a prefix-free set of non-binary variable-length *input codewords*, with a mapping onto a set of variable-length binary *output codewords* [10, p.78-94].
- 16 flush tables that give a mapping from the set of all proper prefixes of *input codewords* onto a set of *output flush words* [10, p.78-94].

Code Index, $i$	Input Symbol Limit, $L_i$	Threshold, $T_i$
0	12	303336
1	10	225404
2	8	166979
3	6	128672
4	6	95597
5	4	69670
6	4	50678
7	4	34898
8	2	23331
9	2	14935
10	2	9282
11	2	5510
12	2	3195
13	2	1928
14	2	1112
15	0	408

Table 2.17: Low-Entropy Code Input Symbol Limit and Threshold

If a *mapped quantizer index*  $\delta_z(t)$  value is already defined to be 'low-entropy' (see equation 2.85), then it shall be encoded using the 'low-entropy' codeword with the largest *code index*  $i$  satisfying the following equation 2.88:

$$\tilde{\Sigma}_z(t) \cdot 2^{14} < \Gamma(t) \cdot T_i \quad (2.88)$$

To start encoding, each 'low-entropy' code has an *active prefix*, which is a sequence of input symbols, and initially it is the null (empty) sequence.

Using the previous *code index*  $i$  and its corresponding *input symbol limit*  $L_i$  (see Table 2.17), equation 2.89 defines every new incoming *input symbol*  $\iota_z(t)$ :

$$\iota_z(t) = \begin{cases} \delta_z(t), & \delta_z(t) \leq L_i \\ X, & \delta_z(t) > L_i \end{cases} \quad (2.89)$$

Then, the *active prefix* for the  $i^{\text{th}}$  low-entropy code is updated by appending the new *input symbol*  $\iota_z(t)$  into the *active prefix*. At this moment, if the *active prefix* matches a complete *input codeword* from the  $i^{\text{th}}$  table [10, p.78-94], the corresponding *output codeword* shall be appended to the bitstream, and the *active prefix* shall be set back to the null sequence one more time.

Moreover, in the specific case that  $\iota_z(t) = X$ , the *residual codeword*  $R'_0(\delta_z(t) - L_i - 1)$  will be appended to the bitstream as well.

#### 2.4.2.2.4 Hybrid Compressed Image Tail

Only when the compression of the image with the *Hybrid Entropy Coder* has completely finished, then the module *Hybrid Compressed Image Tail* attaches an image tail at its very end.

This image tail consists of:

- 16 *flush codewords*, in order of increasing *code index*  $i$ , generated from the 16 remaining *active prefixes* of the 'low-entropy' code.
- The final *high-resolution accumulator*  $\tilde{\Sigma}_z(t)$  value from every *spectral band*  $z$  (so  $N_Z$  values), in order of increasing *spectral band* index, with  $2 + D + \gamma^*$  bits for each.
- A single '1' bit at the end, to let the decoder identify the padding bits added after to fill the *output word size*  $B$ .
- 0-padding bits are appended as needed to reach the next output word boundary.

### 2.4.3 Output packets generation

Once the input is successfully encoded, [10] says nothing about how to output it, but only that it should be put into packets, whose size is a number of bytes controlled by the user-specified parameter *output word size*  $B$ .

Due to the fact that all entropy coders output a variable-length payload, a complex logic is expected to handle not only this packing requirement, but also to make it flexible enough to accept all possible configuration permutations.

This topic is addressed on section 3.9.3.

## 2.5 Differences between Issues 1 and 2

So far only the newer *Issue 2* has been described, which is an update of the *Issue 1*, but not the *Issue 1* itself. To have a graphical idea of their structural differences, Figure 2.5 is upgraded into Figure 2.13:

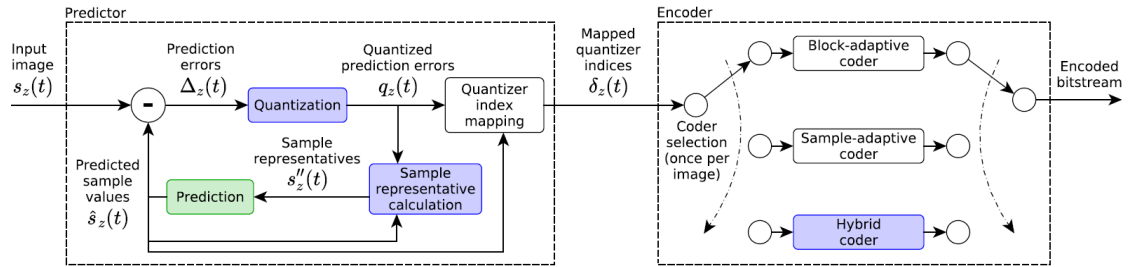


Figure 2.13: Structural differences between Issue 1 and Issue 2 [29, p.3]

White boxes are logic that remain unchanged from *Issue 1* to *Issue 2*, blue boxes are new logic introduced into *Issue 2*, and green boxes are logic that were upgraded from *Issue 1* to *Issue 2*.

Overall, the *Issue 2* upgrades and extends the *Issue 1* to provide not only an effective method for lossless (no error in reconstruction) compression as *Issue 1* does, but also near-lossless (a controlled maximum error in reconstruction) compression of hyper-spectral images [11][10].

Therefore, as Figure 2.13 shows, this near-lossless capability is possible by means of the incorporation of a close-loop quantization scheme in the *Predictor* block, adding the modules *Quantizer* and *Sample Representative*. Moreover, the module *Prediction* is updated to work with the new *sample representative*  $s''_z(t)$  signal.

On the contrary, there are no structural modifications in the *Encoder* part. Modules *Sample-Adaptive Entropy Coder* and *Block-Adaptive Entropy Coder* remain untouched, but instead, a new entropy coding method is included: the *Hybrid Entropy Coder*, which has been designed in order to provide a better compression rate of the low-entropy data under near-lossless compression [10, p.17].

Anyway, this addition also demands an update of the *Encoder Header* [10, p.54].

The logic to pack and output the encoded data (see section 2.4.3) does not change either, as regardless of the chosen entropy coder, all of them output a variable-length signal that must be packed into chunks according to the user configuration.

Fortunately, *Issue 2* has been designed to ensure backwards compatibility with *Issue 1*. In fact, *Issue 1* becomes now a restricted case of the *Issue 2* [10], in other words, *Issue 2* can be configured to be fully compliant with *Issue 1*.

This statement means:

- An image compressed with *Issue 1* can be decompressed with *Issue 2*, only if the second one uses the *Issue 1* configuration.
- An image compressed with *Issue 2* cannot be decompressed with *Issue 1*, unless the first one uses the *Issue 1* configuration.

In terms of the source code structure, the *Issue 1* configuration applied on the *Issue 2* means that the *Predictor* block will work using an open-loop equivalent (being  $s_z''(t) = s_z(t)$ ) and the *Encoder* block shall simply not use the *Hybrid Entropy Coder*.

Table 2.18 enumerates all constraints to impose on *Issue 2* implementation to produce a compressor compliant with *Issue 1*:

Section	Constraint
2.1	Limit dynamic range to $D \leq 16$ bits.
2.4.1.1.1	Do not use supplementary information tables ( $\tau = 0$ ).
2.3.5.1	Do not use narrow local sums.
2.3.2.1	Set the fidelity control method to be lossless.
2.3.4.2	Parameters $\varphi_z = \psi_z = 0$ for all $z$ (so always $s_z''(t) = s_z(t)$ ).
2.3.4	Parameter $\Theta = 0$ (so Sample Repr. subpart on Pred. header not added).
2.3.5.4	All weight exponent offsets $\zeta_z^{(i)} = \zeta_z^* = 0$ .
2.3.5.4	Weight Exp. Offset Flag = '0' (so Weight Exp. Offset Table not added).
2.4.2.1.1	If <i>Sample-Adaptive Entropy Coder</i> used, parameter $\gamma^*$ should not be $>9$ .
2.4.2.2	Do not use the new <i>Hybrid Entropy Coder</i> .

Table 2.18: Constraints to turn Issue 2 into Issue 1 [10]

## 2.6 VUnit framework

VUnit is an open-source unit-testing framework for VHDL and SystemVerilog. It offers the capability to perform continuous and automated testing of HDL code. A biggest point of VUnit is its complementation (not replacement) with traditional testing methodologies by supporting a “test early and often” approach through automation [6].

It supports automatic discovery of test-benches and compilation order to reduce the overhead of testing, and it adds some interesting libraries for common verification tasks. Moreover, it improves the speed of development by supporting incremental compilation and allowing to split up big test-benches into smaller independent tests. It also increases the project’s quality by enabling large regression suites to be run on a continuous integration server too [6].

VUnit is invoked by a user-defined Python script. This file serves as entry point for compiling and running all tests. The framework also provides automatic scanning for all test-benches, automatic determination of compilation order and incremental recompilation of modified sources [4].

The top-level Python script, by default named *run.py*, defines the location for each HDL source file in the project, their associated libraries, external (pre-compiled) libraries and other settings that could be required to compile or simulate the source files [4].

VUnit offers different libraries to simplify the elaboration of test-benches and exchange of data between the DUT and other modules. The most important ones are:

- Run library [3]: Main library of the framework, which allows to execute a VUnit testbench (main process called VHDL test runner) together with the Python-based test runner. It includes a bunch of procedures to start and end such process (*test\_runner\_setup* and *test\_runner\_cleanup*, respectively), to define the different test cases (*run*) and establish a timeout (*test\_runner\_watchdog*), among others.
- Check library [1]: A library that provides different assertions for VHDL in form of check procedures and functions, quite similar to the VHDL *assert* statement.
- Communication library [2]: It provides a high-level asynchronous communication mechanism based on a mathematical model, where actors perform all computation in concurrency with the rest of the system. These actors communicate each other in real time by sending/receiving/replying messages.
- Verification Components library [5]: A collection of entities that implement Rx/Tx standard communication interfaces (UART/AXI/I2C...) to interact with the DUT, and which are controlled from an external process by VUnit messages (back to Communication library).

## 2.7 Logic Synthesis

The *Synthesis* process turns the RTL design (an abstraction of the desired circuit behaviour) into a design made of logic gates and flip flops, totally independent from the hardware target device where the design will run [35].

For a maximum optimization of the *Synthesis* process (which means avoiding sub-design isolation), two main options must be taken into account:

1. Parameter 'flatten\_hierarchy' set to 'rebuilt': It makes to flat the design, perform *Synthesis* and rebuild the original hierarchy again.
2. Disable the *Synthesis Out-Of-Context (OOC)*: This option configures the *Synthesis* with a bottom-up approach, transforming all different Xilinx and CCSDS-123 IPs in an independent way, to put them together at the end.

These two options are enabled by default on the *Xilinx Vivado IDE*, so the *Synthesis OOC* needs to be manually disabled (see section 4.2.2).

As a result, the *Synthesis* process generates a schematic and netlist of the design, along with the power, resources, timing reports. Nevertheless, since this process is totally independent from the hardware platform, these results are estimations that could slightly change in further steps, depending on the FPGA to run the design (see section 2.8).

## 2.8 Implementation

After passing the *Synthesis* process (see section 2.7), the next step is the *Implementation* process, encompassing the following operations [34]:

1. Translate: Merges input netlists and design constraints. The output file describes the logical design reduced to Xilinx primitives (components native to target device).
2. Map: Fits the logic defined from the previous file into FPGA elements. The output file physically represents the design mapped to the components in the Xilinx FPGA.
3. Place and Route: Places and routes the mapped design to the timing constraints. The output file is used as input for the bitstream generation.
4. Generate Programming File: Produces a bitstream file (extension .bit) to be downloaded into the FPGA device.

This bitstream file integrates the design, already optimized to fit into the selected target FPGA, and so, now the updated schematic and power/resources/timing reports can be seen. Unlike in the *Synthesis* process, these files are no longer estimations, but the final versions of them, associated to the specific FPGA to run the design [34].

## 2.9 HDL considerations

This section details some considerations to keep in mind before developing the source code using VHDL, so that no data is lost while being processed (see section 2.9.1) and the source code is indeed synthesizable (see section 2.9.2).

### 2.9.1 VHDL signed vs unsigned signals

As the title of this work already suggests, the implementation of the CCSDS-123 standard here introduced has been carried out with a HDL, more concretely in VHDL. Working with HDLs is noticeably different from programming languages such as C or Java to say some [23], and the present report assumes that the reader is already familiar with both type of languages.

Nonetheless, due to the fact that the algorithm requires to be configurable in terms of data width and data type (*signed/unsigned*), there are a couple of concepts about VHDL signals that are worth mentioning.

First of all, negative numbers can be handled only under *signed* signals, and they use two's complement for that [27], so its dynamic range becomes the half than *unsigned* signals, if speaking in absolute values. That is to say that the same value in bits will be read/interpret as a different number, depending on the chosen data type to work with.

Another major point applies when performing mathematical operations with such signals, and their widths must be controlled all the time, even in intermediate operations. In the worst case (biggest possible numbers), a sum would generate a new value whose width is one more bit than the biggest operand, and in a similar way, a multiplication would generate a new value whose width is the sum of all operand widths.

To overcome these eventualities, so that no data is lost in the process, two different actions must be performed before any mathematical computation:

- All operands are within the same data type (either *signed* or *unsigned*, as no computation can be done under *std\_logic\_vector*).
- All operands, not only the resulting signal, should be resized to the largest possible size first, according to the above-mentioned cases (not more to save resources).

VHDL itself takes care of the *sign extension* while resizing the signals [27], so no need to worry at all whether working with negative numbers (MSb to '1') or not.

Additionally, the *integer* type is not always a good candidate because in VHDL it is limited to 32-bits, so data would be lost when working with bigger numbers. Instead, *std\_logic\_vector*, *signed* or *unsigned* types should be used.

Not following these statements would suppose losing data along the computation chain, and even still having a synthesizable code, so a bug quite difficult to detect later.

Section 10.9 shows several source code examples addressing mathematical equations with this approach.

## 2.9.2 Synthesis design constraints

As already mentioned on section 2.9.1, working with HDLs is a bit different than working with conventional programming languages, so in order to achieve the same result with the same algorithm, a different implementation shall be done when working with either VHDL or C, for instance. Section 2.9.1 is a good example of this.

When a logic has been implemented using HDLs, it must be translated into a physical netlist: a bunch of logic gates, flip flops and other primitives that are used to set a FPGA. This process is called *Synthesis* [35].

This approach of configuring a hardware might limit the implementation of an algorithm if a software approach is used while working. This is indeed not a limitation at all, but simply that a change of mentality is required to implement the algorithm.

Because of the full-configurable nature of the CCSDS-123 standard, two very recurrent problems raised while developing the source code.

The first one is about writing in and reading out a different range of bits of the same signal, depending on the real-time conditions given at a specific moment. Reading out a different range of bits within a signal is not problematic for HDLs, but this is not the case when writing in them. This is not synthesizable, so a workaround must be found.

In such case, a solution is to write in the least significant bits of the signal, and then shifting the data to the left as many positions as needed. And if the signal has other bits that shall not be modified, a XOR gate can be used to keep that data untouched.

In the same way, there are also problems when declaring *for-loops* (static declarations) with conditions that might change over time (dynamic conditions), which the *Synthesis* process will certainly fail. To fix it, it requires a static declaration and an if-statement.

Fortunately, these 'problems' are quite easy to fix. Figures 2.14 and 2.15 show these situations (left) and how to correct them (right), for both cases:

```
if (pntr+b < H) then
  packet(H-b to H-1) := data(U+D-b to U+D-1);
end if;

if (pntr+b < H) then
  packet(0 to b-1) := data(U+D-b to U+D-1);
  packet := packet sll (SIZE_C-b);
end if;
```

Figure 2.14: Non-synthesizable (left) and Synthesizable (right) writing operation

```
for i in pos_x to pos_x+D-1 loop
  do something;
end loop;

subtype range is natural range min to max;
for i in my_loop_range loop
  if (i>=pos_x) and (i<pos_x+D) then
    do something;
  end if;
end loop;
```

Figure 2.15: Non-synthesizable (left) and Synthesizable (right) For-loop



## 3 Design

### 3.1 Overview

Hereafter a proposal for the CCSDS-123 Issues 1 & 2 algorithm is introduced, covering the fully configurable nature that its parameters offers.

The goal of this *Master Thesis* is to provide a fully configurable implementation of the CCSDS-123 standard, to run on a FPGA being the current state-of-the-art (to the author's knowledge), and seeking to be a reference point to any future work related to this standard. Such implementation is a pure PL design, designed using VHDL-2008, in order to take benefit of its configurability features [27].

To begin with, the source code architecture has been designed with the principles of *Modularity*, *Reusability* and *Readability* as its core, so that the design, validation and maintenance phases of the project can be accelerated. Thus, respectively, it is assumed that this source code would need new features or bug fixes over time, some parts of it would be reused in other places, and several engineers would want to have a look at it.

In other words, a design already prepared for the future, and thought to invest a minimum effort to generate maximum results.

Using a top-down approach [12], almost every mathematical equation has been implemented into an individual IP, most of them just requiring a single clock cycle to generate a valid output, given a valid input. To ensure a proper synchronization among all IPs, all of them receive in (and forward out too) an *enable* signal and the current *image coordinates* of the incoming hyperspectral image, so that the complete design can be stopped at any moment if desired so, and every IP knows with which particular pixel of the image is working at any moment.

Moreover, every signal within the design has a default value at a declaration time as well as when the reset condition is met. This makes the design more robust since no unexpected values can be generated, avoiding open-circuit and short-circuit states.

Finally, this design comes together with a TCL framework that automates the creation of a *Xilinx Vivado IDE* project, the definition of the target device, the addition of source code and constraint files, and the bitstream generation. This makes the design be just a couple of clicks away to flash it into a FPGA and work with it in the real world.

Overall, a functional implementation of the CCSDS-123 standard designed to be very efficient from the very beginning, automating all its side-work, and prepared to the unknowns of the future.

Section 3.1.1 shows the timing diagram of the complete implementation.

### 3.1.1 Timing diagram

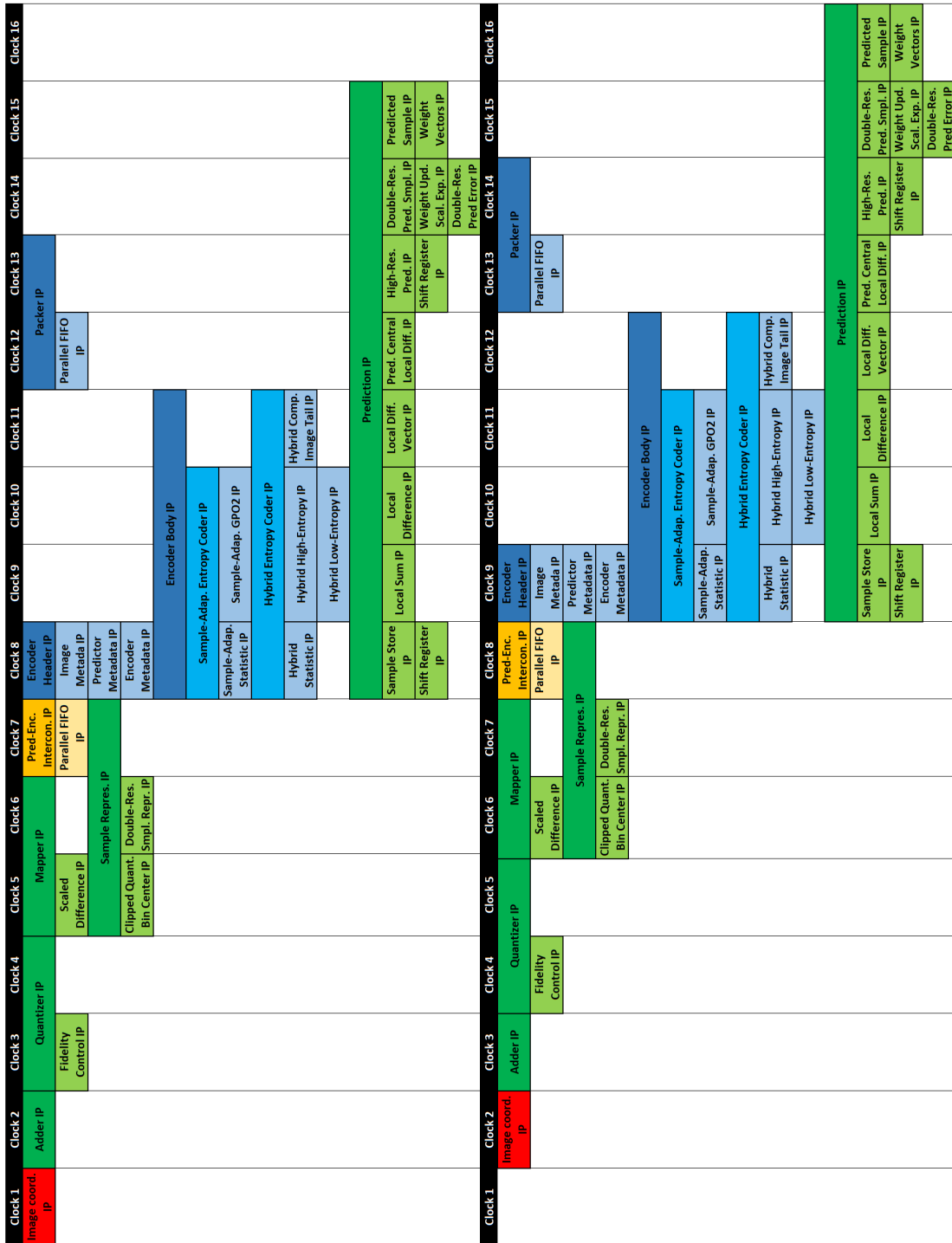


Figure 3.1: CCSDS-123 Top Entity IP timing diagram

## 3.2 Development tools

The algorithm here presented has been fully developed with the software tool Xilinx Vivado Design Suite 2019.1: used for RTL design, synthesis, implementation, constraints definition, bitstream and power/utilization/timing reports generation. No Vivado IP integrator option was used for code design, but simply plain text HDL files, so that the source code could easily be exported to other ASIC vendor tools, if necessary.

For simulation and debugging instead, the software tool ModelSim Starter Edition 2020.1 Engine has been used because of a higher compatibility with HDLs and way better performance [22].

The chosen HDL has been VHDL-2008 (the last compatible revision with most ASIC tools). Unlike Verilog, VHDL is a strongly typed HDL (and more verbose), and it is taught and used way more in Europe rather than Verilog, so it is easier to find information about this HDL as well as to consult with other developers.

Moreover, with the aim of a smoother design flow and better code management, the complete project workspace is controlled through a TCL-script based framework, which mixes both Vivado-specific and general TCL commands (which means it is executed inside the Vivado IDE TCL interpreter). This framework automates all steps involved in the project development: Vivado project creation and configuration, sources/constraint files and libraries association, Xilinx's IPs integration, bitstream/HW description file/reports generation, etc.

Refer to section 4.2 to see more information about this TCL framework and the rest of the tools.

The knowledge of how to use all these tools and HW/SW programming languages are assumed to be familiar for the reader.

## 3.3 Hardware platform

As described on chapter 1, the NTNU SmallSat project integrates a SoC (uC + FPGA) as the current standard choice in small-satellite missions because of its reconfigurable nature and the possibility to execute complex tasks in parallel, so a tight integration between hardware and software. More concretely for this work, the *Zynq UltraScale+ MPSoC ZCU102 Evaluation Board* is the chosen one.

The Zynq UltraScale+ MPSoC family is based on the Xilinx UltraScale MPSoC architecture, whose block diagram is depicted on Figure 3.2. This architecture combines uC Processing System (PS) (quad-core ARM Cortex-A53 and dual-core ARM Cortex-R5F) and a FPGA Programmable Logic (PL) UltraScale architecture in a single device:

The *Zynq UltraScale+ XCZU9EG-2FFVB1156E MPSoC*.

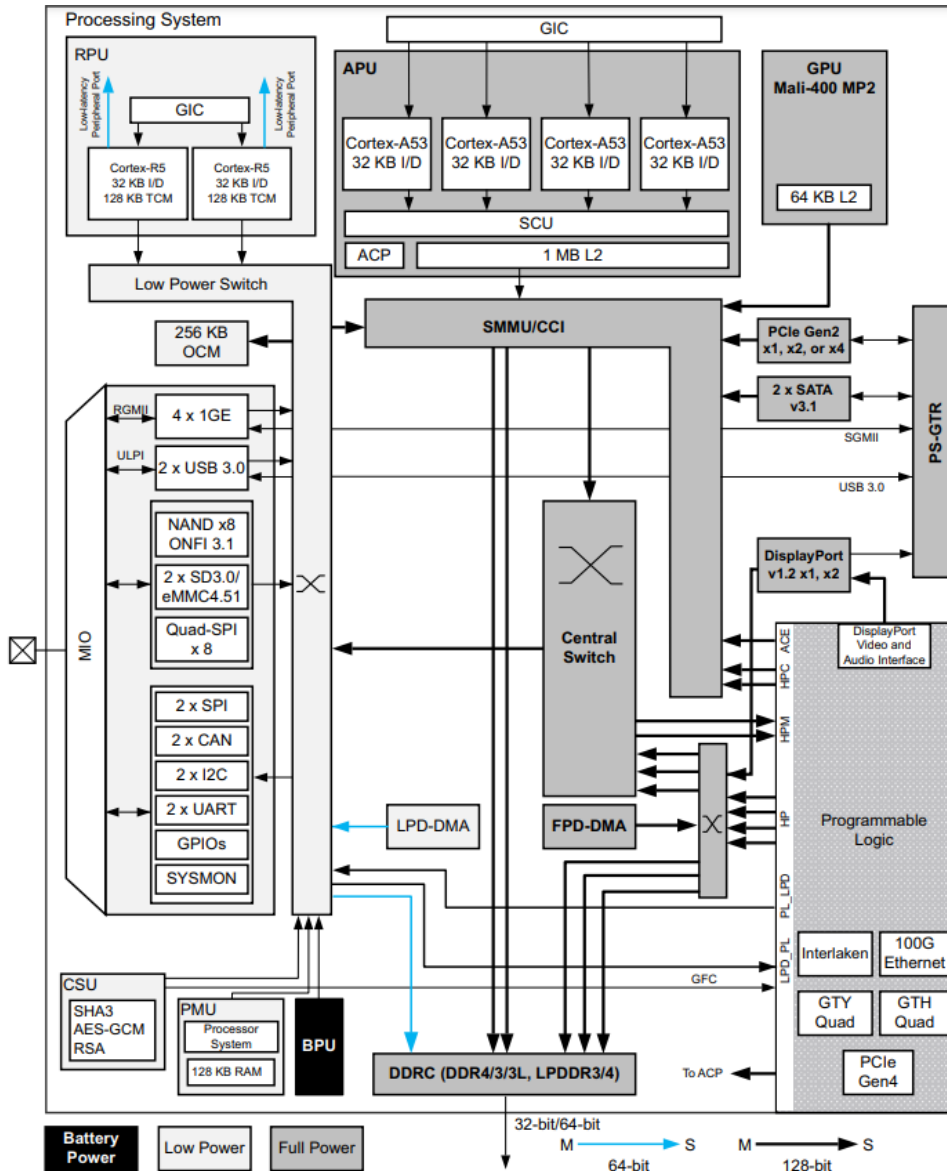


Figure 3.2: Xilinx UltraScale MPSoC architecture

The algorithm here introduced is a pure 100% FPGA (PL) design, so there is no need to go into more details with this hardware platform. Anyway, if desired, check out references [33] and [37] for more information about it.

Section 5.2.2 details the primitive resources (basic components) that this FPGA include, compared with how many of them were used to implement the presented design.

Additionally, Figure 10.1 shows the package pinout of this SoC, where the pins to use for the constraint files were extracted.

## 3.4 Source code architecture

The source code development is carried out with 3 clearly differentiated principles:

1. Modularity: The possibility to add and/or remove IPs into/from the design with a minimum effort.
2. Reusability: Each IP shall implement the minimum necessary logic, so they can be used on many places, and the union of 'simple' IPs can create more complex IPs.
3. Readability: The source code should be simply enough to understand it (ideally) at first glance and to have a 1:1 match with the provided documentation.

The aforementioned points offer a bunch of advantages, already thinking in the near and distant-future (after the presentation of the current paper), and indeed they are totally independent from the algorithm itself to implement.

To begin with, these principles ensure a resulting source code with a high-level of scalability and maintainability, something mandatory due to the fact that the CCSDS-123 algorithm might be updated in the future once more with new *issues*. Additionally, being the source code quite understandable for other developers, they could easily help to fix/improve it as well. Finally, this approach accelerates the development and verification phases of the source code, which means much more time can be invested in the algorithm itself, rather than the architecture framework, for example.

To sum up, all this is translated into a big project where potentially all interested engineers can take part on helping improve the outcome.

Finally, it must be mentioned that every *signal* and *variable* in the design has been given a default value, not only when declared, but also when the reset condition is met, and that they are written from one unique place at a time. This point ensures there will be neither open- nor short-circuits at any place under any condition in the whole design.

Sections 10.5 and 10.6 detail more specifically all guidelines (VHDL style guide and coding guidelines) used for creating the source code.

### 3.4.1 Packages

According to the guidelines stated on section 3.4, some VHDL packages have been created to integrate constants and other static declarations together, so that there is no need anymore to constantly see duplicated elements in the code that make it dirty.

Thanks to these packages, the different IPs will only include dynamic code, or functional statements.

In order to make them modular as well, there is a group of packages for every main block of the algorithm: *Image coordinates* block or general (section 3.6), *Predictor* block (section 3.7) and *Encoder* block (section 3.9).

Section 10.10 shows one example of VHDL package used.

Every 'main-block' group of packages include:

- Parameters: Static configuration of every block. Such parameters are listed on sections 10.2, 10.3 and 10.4.
- Types: General and custom VHDL types/records/arrays declaration. Most of them are unconstrained (VHDL 2008 feature [27]), so that the same declaration can be used with any parameters configuration, being the array widths the most palpable case.
- Utils: General and specific functions declarations and their implementations. Similar to the *Types* packages, these functions are also implemented using unconstrained vectors, to ensure they will work for any parameters configuration.
- Components: Component declaration of every IP, letting all IPs access their neighbours, if required.
- Others: Very specific constants (basically tables) that are fairly long and not easy to read. Examples are the *Supplementary Information Tables* (section 3.9.1.1.1) and *Hybrid Code/Flush Tables* [10, p.78-94].

Thus, isolating all this code into independent packages makes the source code way more readable, and saves hundreds (if not thousands) lines of code within the IPs themselves by simply calling these packages instead. Thus, and as said before, with this approach the IPs will contain only the specific part of the algorithm they are pretending to implement.

In the same way, it also ensures that no code is duplicated, which avoids confusion and saves tons of time while developing and debugging.

### 3.4.2 Block diagrams description

Sections 3.5 to 3.9 explain the design of the complete CCSDS-123 Issues 1 & 2, and several block diagrams are used as support for the most complex ones. Therefore, they have been standardized to give as much information as possible on a very small and minimalist drawing.

Figure 3.3 is an example block diagram, and the following statements can be extracted from reading it:

1. Data-flow always moves from left side (input signals) to right side (output signals).
2. Signals have an identification name near them, with the same colour as its arrow.
3. All blocks also have their specific colour:
  - Light grey boxes are the (relative speaking) top IPs, with a short identification name on their top-center part.
  - Black boxes are sub-IPs, with a short identification name inside.
  - Golden boxes are processes, regardless of its identification name inside.

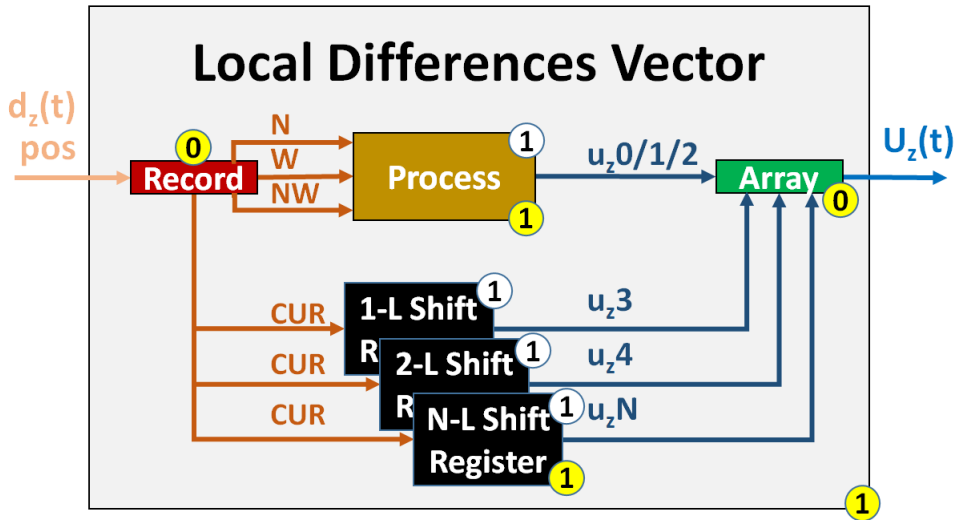


Figure 3.3: Block diagram example

- Maroon boxes are record types (custom arrays), where a signal is either compressed or uncompressed, always with name 'Record' inside.
  - Green boxes are array types, where a signal is either compressed or uncompressed, always with name 'Array' inside.
  - Blue trapezoids (not displayed in Figure 3.3) are either multiplexers or demultiplexers, with their selection signals always connected on one side.
4. All boxes (even the top IP) have a yellow label on their right-bottom corner, showing the required clock cycles to produce a valid output, given a valid input.
    - Value '0' means that such box only implements conditional logic.
    - Value '>0' means that such box at least implements sequential logic.
  5. All boxes (except the top IP) have a white label on their right-up corner, showing its execution order from top IP perspective.
    - Different boxes with the same white label number means that they are executed in parallel.
    - Different boxes with different white label numbers means that they are executed in series.

Apart from this, it is important to highlight that there are two things not represented in these diagrams, simply for the sake of readability:

- 'Enable' and 'Image coordinates' signals are not displayed here. Exceptions are the *Top Entity IP*, *Image Coordinates IP* and *Encoder Header IP*.
- Signals that require a relative delay (inputs connecting directly to an intermediate IP/process) do not show such delay, but it is understood from the connection itself.

### 3.5 CCSDS-123-Issue2 Top Entity IP

The very top entity of the CCSDS-123 algorithm (called Issue 2 for simplicity, but it also covers the Issue 1) shows its architecture on Figure 3.4:

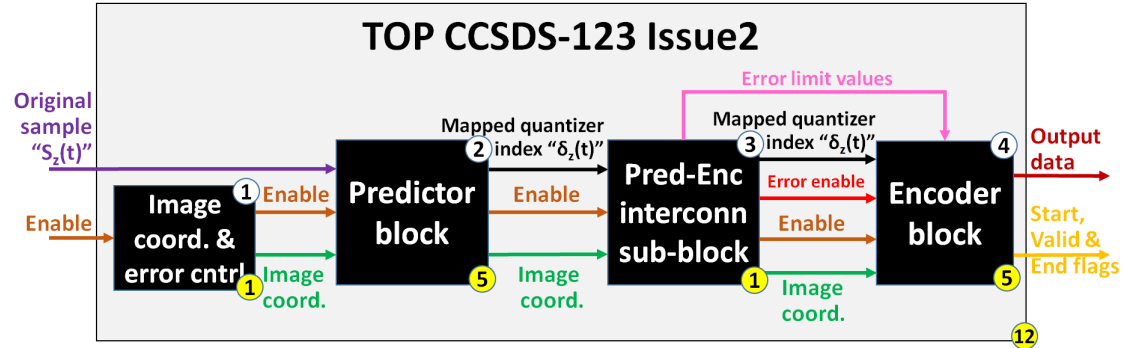


Figure 3.4: Top entity IP block diagram

There are 4 major sub-IPs connected on cascade, with same execution order as listed hereafter:

1. Image Coordinates Control IP: It keeps track of the incoming samples by counting them, taking into consideration if they come in with a BSQ, BIL or BIP order: See section 3.6.
2. Predictor IP: It computes prediction of the next samples by reading neighbour samples, and it applies an acceptable error on it, if configured so: See section 3.7.
3. Predictor-Encoder Interconnection IP: Only instantiated if *Periodic Error Limit Updating* option is enabled (see section 3.7.2.1.1), it fetches *Predictor IP*'s result and outputs it, or the *Error Limit Values*, to the *Encoder IP*: See section 3.8.
4. Encoder IP: It encodes the incoming data with the selected Entropy coder (*Sample adaptive coder* [section 3.9.2.1] or *Hybrid coder* [section 3.9.2.2] <sup>1</sup>): See section 3.9.

This very top IP integrates all defined configuration constraints too:

- Local sum config. moves to *column-oriented* if image has width  $N_X = 1$ , or to *neighbour-oriented* if working under *full prediction mode* (see section 2.3.5.1).
- *Full prediction mode* cannot be used if image has width  $N_X = 1$  (see section 2.3.5).
- *Periodic Error Limit updating* cannot be used with BSQ order (see section 2.3.2.1.1)

The execution order of these blocks is from left to right, just as the numbered white labels list (see section 3.4.2). In the same way, as its numbered yellow label shows, the *Top entity IP* needs a total of 12 clock cycles (basically the sum of all blocks execution time) to produce a valid output, given a valid input.

<sup>1</sup>*Block-Adaptive Entropy Coder* is not mentioned here, because it was not implemented.



Nevertheless, the previous number does not match with the quantity of clock cycles to (de)compress an input image. See section 5.4 for more detailed information.

As explained on section 2.1, the source code shows that each *original sample*  $s_z(t)$  value has a range or number of bits defined by equation 2.2, but some IPs here below use signals with more bits than that. Unless the documentation says explicitly to use another quantity, this is done to ensure no data is lost when intermediate computed values are bigger than expected (see section 2.9.1).

### 3.5.1 Top Entity IP configuration

Apart from the VHDL packages created to hold the parameters of the system (see section 3.4.1), the declaration of the very *Top Entity IP* also includes a bunch of extra configuration elements in its *Generic* part [16].

The difference between the parameters from the packages and the elements from the *Generic* part is, that the first group configures the mathematical operations themselves to be executed along the whole algorithm, and the second group are conditional statements defining which parts of the algorithm should be whether enabled or disabled.

These configuration elements are listed below:

- **SMPL\_TYPE\_G**: It defines the input samples data type (0: *signed* type samples, 1: *unsigned* type samples).
- **SMPL\_ORDER\_G**: : It declares the order type of the input samples (00: BSQ order, 01: BIP order, 10: BIL order).
- **HEADER\_EN\_G**: It defines whether the *Encoder Header* is enabled or not (0: Disabled, 1: Enabled).
- **PREDICT\_MODE\_G** It defines the working *Prediction mode* (1: Full prediction mode, 0: Reduced prediction mode).
- **LSUM\_TYPE\_G**: It declares the *local sum* calculation type (00: Wide neighbour, 01: Narrow neighbour, 10: Wide column, 11: Narrow column).
- **W\_INIT\_TYPE\_G**: It declares the *weights initialization* type (1: Custom weights initialization, 0: Default weights initialization).
- **PER\_ERR\_LIM\_UPD\_G**: It defines whether the *Periodic Error Limit Updating* option is enabled or not (0: Disabled, 1: Enabled).
- **FIDEL\_CTRL\_TYPE\_G**: It declares the accepting maximum error during compression (00: lossless, 01: absolute error limit only, 10: relative error limit only, 11: both absolute and relative error limits).

- **ABS\_ERR\_BAND\_TYPE\_G**: It defines the absolute error limit values type (1: band-dependent, 0: band-independent).
- **REL\_ERR\_BAND\_TYPE\_G**: It defines the relative error limit values type (1: band-dependent, 0: band-independent).
- **ENCODER\_TYPE\_G**: It selects the *Entropy Coder* type (00: Sample-Adaptive Entropy, 01: Hybrid Entropy).
- **W\_INIT\_TABL\_FLAG\_G**: It defines whether the *Weight Initialization Table* is enabled or not inside the *Encoder Header* (0: Disabled, 1: Enabled).
- **W\_EXP\_OFF\_TABL\_FLAG\_G**: It defines whether the *Weight Exponent Offset Table* is enabled or not inside the *Encoder Header* (0: Disabled, 1: Enabled).
- **DAMP\_TABLE\_FLAG\_G**: It defines whether the *Damping Table* is enabled or not inside the *Encoder Header* (0: Disabled, 1: Enabled).
- **OFFSET\_TABLE\_FLAG\_G**: It defines whether the *Offset Table* is enabled or not inside the *Encoder Header* (0: Disabled, 1: Enabled).
- **ACCU\_INIT\_TABLE\_FLAG\_G**: It defines whether the *Accumulator Initialization Table* is enabled or not inside the *Encoder Header* (0: Disabled, 1: Enabled).
- **RESTRICT\_CODE\_G**: It defines whether the *Restricted set of code options* is used or not (0: NOT used, 1: used).
- **UDEF\_DATA\_G**: It declares the *User Defined Data* field for the *Supplementary Information Tables* (any number of 8-bits accepted).
- **SUPL\_TABLE\_TYPE\_G**: It defines the elements *Data Type* within the 15 *Supplementary Information Tables* (00: unsigned integer, 01: signed integer, 10: float).
- **SUPL\_TABLE\_PURPOSE\_G**: It defines the *Purpose* of the 15 *Supplementary Information Tables* (from 0 to 15, defined on Table 2.4).
- **SUPL\_TABLE\_STRUCT\_G**: It defines the *Structure* of the 15 *Supplementary Information Tables* (00: zero-dimensional, 01: one-dimensional, 10: two-dimensional-zx, 11: two-dimensional-yx).
- **SUPL\_TABLE\_UDATA\_G**: It defines the *Supplementary User-Defined Data* of the 15 *Supplementary Information Tables* (any number of 4-bits is accepted).
- **ENDIANNESS\_G**: It defines the *Endianness* of the output packets with the compressed data (0: Little Endian order, 1: Big Endian order).

All these elements are also discussed in the sections below, appearing when required to support some explanations.

### 3.6 Image Coordinates Control IP

Figure 3.5 is the block diagram of the *Image Coordinates Control IP*, the first one in the chain. This IP is not defined on the documentation itself, but anyway implemented as a solution to control the coordinates of the incoming image, sample by sample, through the complete design.

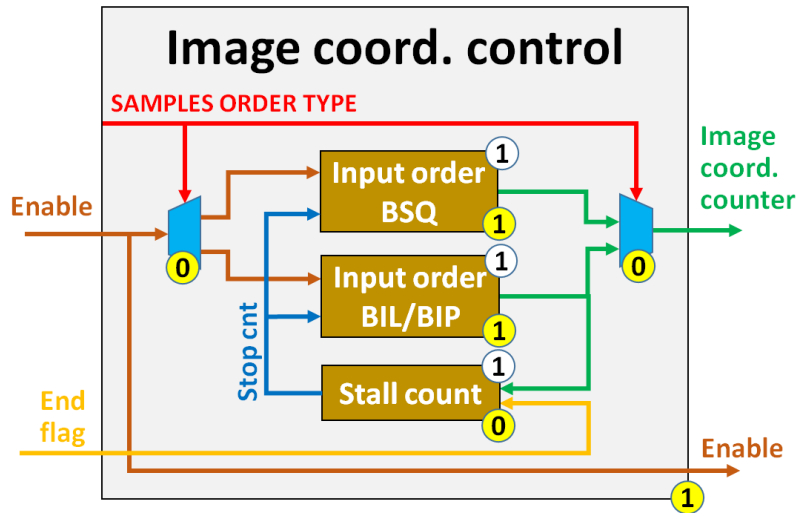


Figure 3.5: Image coordinates IP block diagram

When the incoming *enable* signal is asserted for the first time, an internal counter is increased '+1' on every clock cycle.

Such increase depends on the configured samples input order, which instantiates either a clocked process for the BSQ order (Figure 2.2), or another clocked process for BIL/BIP order (Figure 2.3), together with a multiplexer and demultiplexer (combinational logic) to do the proper connections of the selected process to the outside world.

Moreover, there is another process that prevents the IP processing a new image (or counting from the beginning again) until the the current one has been successfully (de)compressed, using the *end flag* input signal (see section 3.9.3).

The outputs are the image counter, with x, y, z and t components (equations 2.1 and 2.6) and *enable* signal, delayed one clock cycle to be used for the next IPs in the chain.

This IP just needs one clock cycle to produce a valid output, given a valid input. Section 10.11 shows part of its source code.

### 3.7 Predictor Top IP

The *Predictor IP* uses an adaptive linear prediction method to predict the value of each image sample based on the nearby samples in a small neighborhood. Figure 3.6 shows its block diagram, the most complex one if sub-blocks are taken into account:

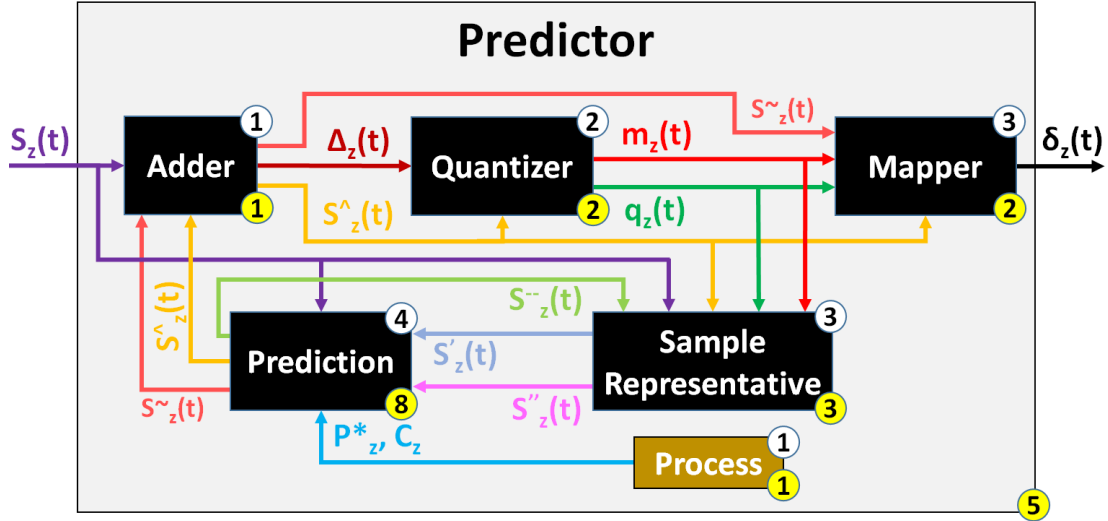


Figure 3.6: Predictor IP block diagram

It is interesting to realize that the *Predictor* block diagram from Figure 3.6 shows more dependencies among its sub-IPs than in Figure 2.5, provided on documentation [10, p.18], fruit of analyzing all involved equations.

As one can see on Figure 3.4, the *Predictor* IP is essentially an interconnection point of its sub-IPs, without adding any extra logic.

The execution starts with the *Adder IP*, producing the *prediction residual*  $\Delta_z(t)$  and forwarding the *predicted sample*  $\hat{s}_z(t)$  values. These values are taken by the *Quantizer IP* to compute the *quantized prediction residual*  $q_z(t)$  and *maximum error*  $m_z(t)$  values, feeding the *Mapper IP* and the *Sample Representative IP* at a time.

On one side, the *Mapper IP* uses these signals to produce the *mapped quantizer index*  $\delta_z(t)$  value, the final output of the *Predictor* block. On the other side, the close-loop starts with the *Sample Representative IP*, using the same signals to compute the *sample representative*  $s''_z(t)$  and *clipped quantizer bin center*  $s'_z(t)$  values, which stimulate the *Prediction IP* to compute the *predicted sample*  $\hat{s}_z(t)$ , *high-resolution predicted sample*  $\check{s}_z(t)$  and *double-resolution sample representative*  $\tilde{s}_z(t)$  values, and back again to the *Adder IP* to finish the close-loop.

The *enable* and *image coordinates* signals are forwarded to every single component in the design, so that all IPs know which image sample are working with at any moment, but they are not displayed in Figure 3.4 for the sake of readability. Indeed, they are used in a clocked process to keep track of the *preceding spectral bands*  $P_z^*$  (equation 2.32) too

Taking into account the open-loop branch, the *Predictor IP* just needs 5 clock cycles to generate a valid output given a valid input.

Nevertheless, this is different from the number of clock cycles to produce the final *mapped quantizer index*  $\delta_z(t)$  values. See section 5.4 for more detailed information.

### 3.7.1 Adder IP

Although the *Adder IP* is not complex, but it is very important to understand what it does, as its identification name just states half of it. Figure 3.7 is its block diagram:

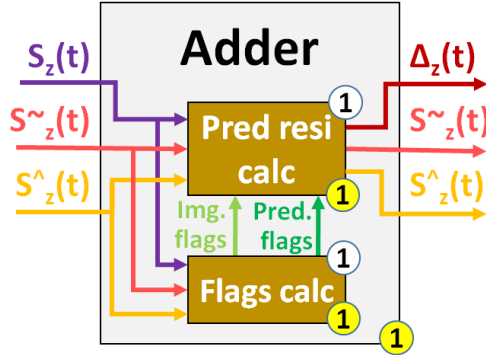


Figure 3.7: Adder IP block diagram

The main task is a clocked process to compute the *prediction residual*  $\Delta_z(t)$ : the difference between the *predicted sample*  $\hat{s}_z(t)$  and *original sample*  $s_z(t)$  values, according to equation 2.7. The same process also forwards the *predicted sample*  $\hat{s}_z(t)$  and *double-resolution sample representative*  $\tilde{s}_z(t)$  values for further calculations in the next IPs.

Apart from that, this IP is also in charge of merging the close-loop branch back to the main line, but as such loop obviously generates the *predicted sample*  $\hat{s}_z(t)$  value at a certain time later than when the *original sample*  $s_z(t)$  value was received, a second clocked process is required to take care of the synchronization of these two signals.

At the very beginning, the *prediction residual*  $\Delta_z(t)$  is computed to stimulate the system and to generate the *predicted sample*  $\hat{s}_z(t)$  values, and at the same time, the *predicted sample*  $\hat{s}_z(t)$  and *original sample*  $s_z(t)$  values are being stored within FFs.

When all *original sample*  $s_z(t)$  values are forwarded, the *image flag* is asserted, and once all *predicted sample*  $\hat{s}_z(t)$  values are received, the *prediction flag* is asserted too. At this very moment, the *prediction residual*  $\Delta_z(t)$  is computed once again, but now the *Adder IP* can compute it synchronized, as equation 2.7 defines.

Additionally, this IP also includes a configurable parameter to define if the incoming *original sample*  $s_z(t)$  is either *signed* or *unsigned*, so that values are properly read.

This sign configuration applies only to this very IP, as any other signal inside the *Predictor IP* block can potentially assume negative values, being this is the reason why the other signals must be *signed* and with a bigger number of bits than *original sample*  $s_z(t)$  value, so that no data is lost.

Thus, although these 2 clocked processes in parallel make *Adder IP* require just 1 clock cycle to produce a valid output, given an input, the real *prediction residual*  $\Delta_z(t)$  is not generated after forwarding the whole input image once ( $N_X * N_Y * N_Z$  samples). See section 5.4 for more detailed information. Section 10.12 shows part of its source code.

### 3.7.2 Quantizer IP

Figure 3.8 shows the block diagram of the *Quantizer IP*:

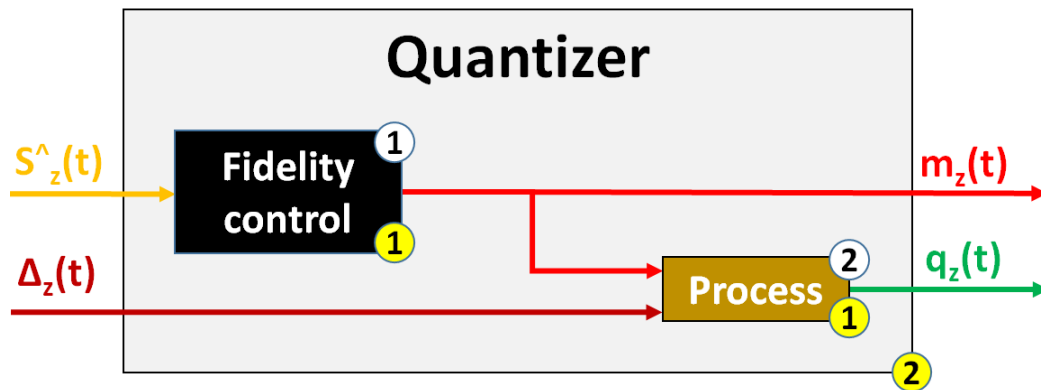


Figure 3.8: Quantizer IP block diagram

The *Quantizer IP* is responsible for the following operations, described in the same order as executed:

1. First, the *Fidelity Control IP* uses the incoming *predicted sample*  $\hat{s}_z(t)$  value to generate the *maximum error*  $m_z(t)$  value, determined via the *Error limit values* and user-specified settings to know the error type (refer to section 3.7.2.1).
2. Then, on a clocked process, the *prediction residual*  $\Delta_z(t)$  (delayed one clock cycle to be synchronized with the previous operation), is quantized using a uniform quantizer with a step size  $2m_z(t) + 1$  to generate the *signed quantizer index*  $q_z(t)$ , as equation 2.8 shows. This single task is separated into several variables to make it easier to understand and debug.

These two clocked operations take one clock cycle each to execute, and they are executed sequentially, so a total of 2 clock cycles are required for the *Quantizer IP* to produce a valid output, given a valid input.

#### 3.7.2.1 Fidelity Control IP

The *Fidelity Control IP*, shown in Figure 3.8, computes the *maximum error*  $m_z(t)$  value, based on the incoming *predicted sample*  $\hat{s}_z(t)$  and *Error limit values*. Its operations are:

1. Combinational logic is used to read both *absolute and relative error limit values* and to pass them to the next process, extracted from the *Error limit values* table (see section 3.7.2.1.1). The different possibilities are:
  - Lossless compression: Both *absolute and relative error limit values* are fixed to zero.

- No lossless compression and *Periodic Error Limit Updating* option disabled: The first position of both absolute and relative error limit values arrays are always taken.
  - No lossless compression and *Periodic Error Limit Updating* option enabled: The arrays position are continuously monitored, and the right absolute and relative error limit values are extracted from the arrays.
2. The *maximum error*  $m_z(t)$  value is computed on a clocked process by means of the *predicted sample*  $\hat{s}_z(t)$  value and the user-specified error configuration. The different possibilities are:
- Lossless compression: *maximum error*  $m_z(t)$  value is simply fixed to 0 (see equation 2.9).
  - Absolute error limit: *maximum error*  $m_z(t)$  value is computed according to equation 2.10.
  - Relative error limit: *maximum error*  $m_z(t)$  value is computed according to equation 2.12).
  - Both absolute and relative error limits together: *maximum error*  $m_z(t)$  value is the minimum one between absolute and relative error limit cases (see equation 2.14).

Figure 3.9 shows the block diagram of the *Fidelity Control IP*:

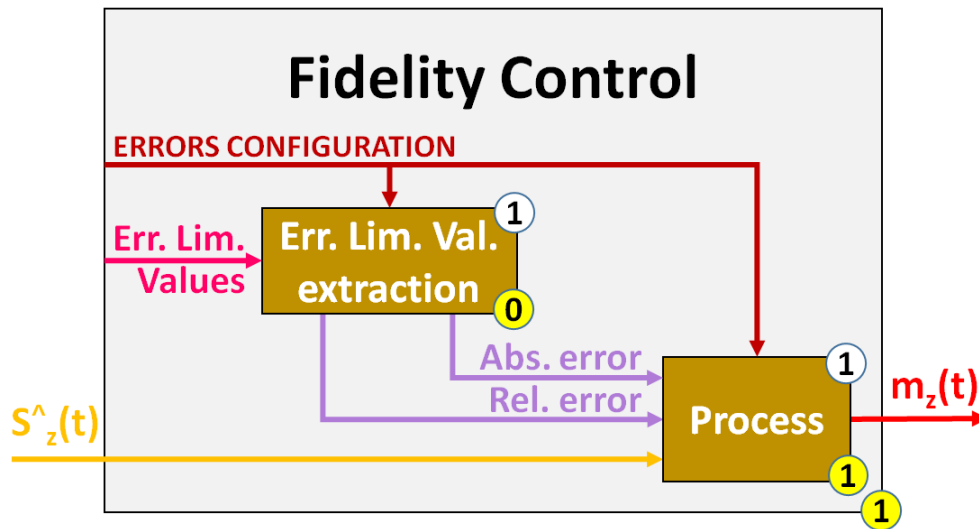


Figure 3.9: Fidelity Control IP block diagram

Just as this figure shows, the mix of combination logic and sequential logic in parallel defines a total of 1 clock cycle to produce a valid output, given a valid input.

### 3.7.2.1.1 Error Limit Values Table

User-specified *absolute and/or relative error limit values* are used to control the *maximum error*  $m_z(t)$  value for each sample (see section 3.7.2.1), and they are not computed at all, but just defined by the user itself.

Depending on the errors configuration, *absolute and relative error limit values* can be given in two different formats:

- *Band-independent*: A unique value for all *spectral bands*. Applicable to both *absolute and relative error limit values*.
- *Band-dependent*: A different value for each *spectral band* ( $N_z$ -long array). Applicable to both *absolute and relative error limit values*.

Moreover, there is the '*Periodic Error Limit updating*' option (disabled for BSQ input order), which offers the possibility to update the *Error limit values* after a configurable number of image frames (see section 2.3.2.1.1). In order words, an array of the previous *Error limit values* types.

To implement such requirements, a VHDL package that covers all possibilities is defined here: a custom array of size  $U$  that includes inside:

- Absolute error limit constant  $A^*$  (when *absolute band-independent* configured).
- Absolute error limit array  $a_z$ , with size  $N_z$  (when *absolute band-dependent* configured).
- Relative error limit constant  $R^*$  (when *relative band-independent* configured).
- Relative error limit array  $r_z$ , with size  $N_z$  (when *relative band-dependent* configured).

By default, such big array is initialized on every position with the same values for every sub-type. The user can modify them as desired/required.

As explained on section 3.7.2.1, the size  $U$  depends on the *image coordinates*, and as all IPs have access to these coordinates, the *Fidelity Control IP* is able to take the proper array position at any time.

This VHDL package is static data, so there is no delay to generate it, and the output values can be used immediately.



### 3.7.3 Mapper IP

Figure 3.10 shows the block diagram of the *Mapper IP*:

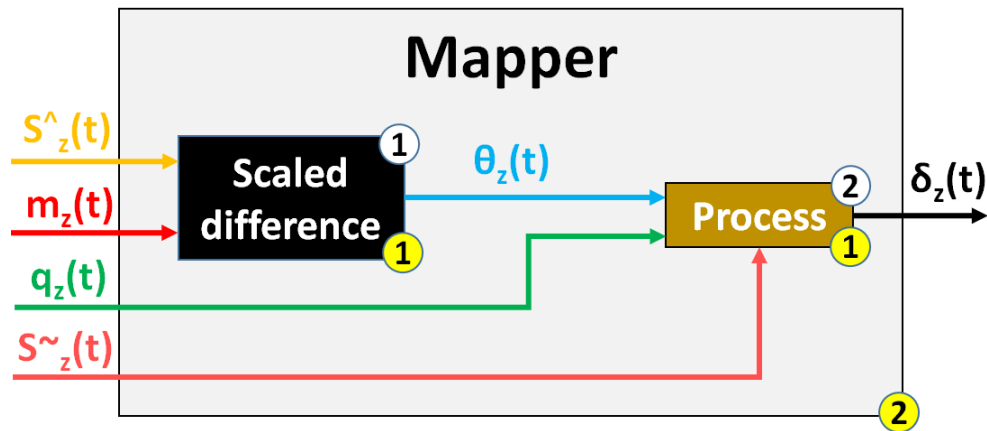


Figure 3.10: Mapper IP block diagram

This IP is in charge of mapping the *signed quantizer index*  $q_z(t)$  values with a configured step size. Its operations, executed in the same order as explained, are:

1. The *Scaled difference* IP uses the incoming *predicted sample*  $\hat{s}_z(t)$  and *maximum error*  $m_z(t)$  values to compute the *scaled difference*  $\theta_z(t)$  value (see section 3.7.3.1).
2. The *quantizer residual*  $q_z(t)$  and *double-resolution predicted sample*  $\tilde{s}_z(t)$  values, delayed both of them one clock cycle to be synchronized, are used along with the *scaled difference*  $\theta_z(t)$  value on a clocked process to compute the *mapped quantizer index*  $\delta_z(t)$  value, just as equation 2.16 shows.

The equation is computed with different variables for the sake of readability and debugging purposes.

The sequential sub-IP and clocked process define a total of 2 clock cycles to produce a valid output, given a valid input.

#### 3.7.3.1 Scaled Difference IP

The *Scaled Difference IP*, shown in Figure 3.10, uses the incoming *predicted sample*  $\hat{s}_z(t)$  and *maximum error*  $m_z(t)$  values on a clocked process to generate the *scaled difference*  $\theta_z(t)$  value, according to equation 2.17.

This operation is broken down with different variables for the sake of readability and debugging purposes.

This single task just requires 1 clock cycle to generate a valid output, given a valid input. Section 10.13 shows part of its source code.

### 3.7.4 Sample Representative IP

Figure 3.11 shows the block diagram of the *Sample Representative IP*:

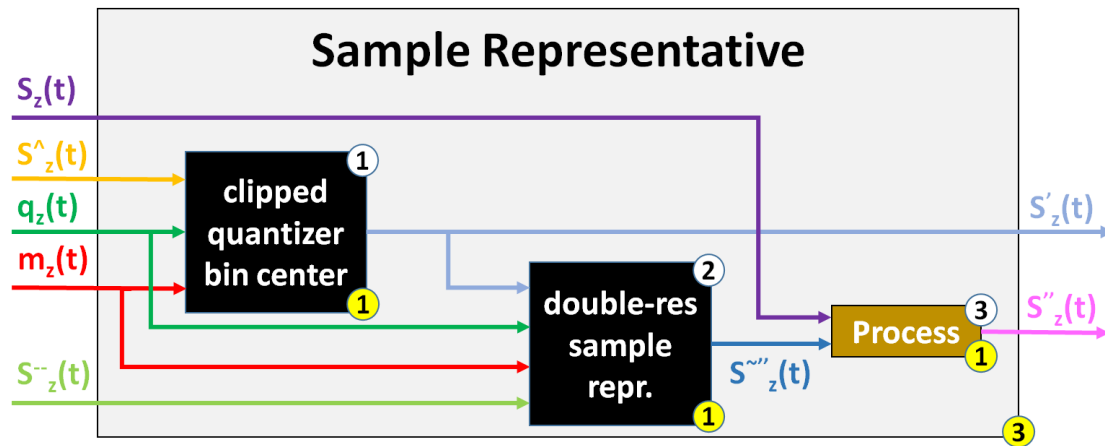


Figure 3.11: Sample Representative IP block diagram

This IP implements the following operations, executed in the same order as explained:

1. First action is the computation of the *clipped quantizer bin center*  $s'_z(t)$  value (see section 3.7.4.1). Even though this value is used by the next IP here, it is also delayed two clock cycles to be outputted for other IPs as well.
2. Using the previous output value along with the *maximum error*  $m_z(t)$ , *quantizer index*  $q_z(t)$  and *high-resolution predicted sample*  $\check{s}_z(t)$  values (the three of them delayed one clock cycle to synchronize with  $s'_z(t)$ ), the *double-resolution sample representative*  $\tilde{s}_z(t)$  value is computed (see section 3.7.4.2).
3. Last but not least, the *double-resolution sample representative*  $\tilde{s}_z(t)$  is used along with the *original sample*  $s_z(t)$  (in this case delayed two clock cycles to be synchronized with the previous operation output) on a clocked process to produce the *sample representative*  $s''_z(t)$  value, just as equation 2.18 states.

These three tasks are executed in series, so a total of 3 clock cycles are required to produce a valid output, given a valid input.

#### 3.7.4.1 Clipped Quantizer Bin Center IP

The *Clipped Quantizer Bin Center IP*, shown in Figure 3.11, uses a clocked process to compute the *clipped quantizer bin center*  $s'_z(t)$  value by using the *maximum error*  $m_z(t)$ , *quantizer index*  $q_z(t)$  and *high-resolution predicted sample*  $\check{s}_z(t)$  values, according to equation 2.19.

This operation is broken down with different variables for the sake of readability and debugging purposes.

This IP requires only 1 clock cycle to produce a valid output, given a valid input.

### 3.7.4.2 Double-Resolution Sample Representative IP

Figure 3.12 is the block diagram of the *Double-Resolution Sample Representative IP*:

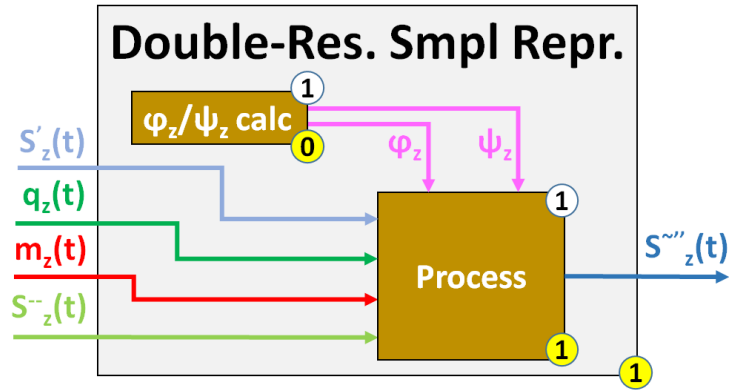


Figure 3.12: Double-Resolution Sample Representative IP block diagram

This IP uses a clocked process to compute the *double-resolution sample representative*  $\tilde{s}_z(t)$  value by using the *clipped quantizer bin center*  $s'_z(t)$ , *maximum error*  $m_z(t)$ , *quantizer index*  $q_z(t)$  and *high-resolution predicted sample*  $\check{s}_z(t)$  values, as equation 2.20 states, broken down into several steps to make it easier to understand and debug.

Prior to this computation, combinational logic is used to extract the right value from parameters *damping*  $\varphi_z$  and *offset*  $\psi_z$ , a new value per new *spectral band*  $z$  (see equations 2.22 and 2.23), fixing *offset*  $\psi_z$  to 0 in case lossless compression is configured.

This IP requires only 1 clock cycle to produce a valid output, given a valid input.

### 3.7.5 Prediction IP

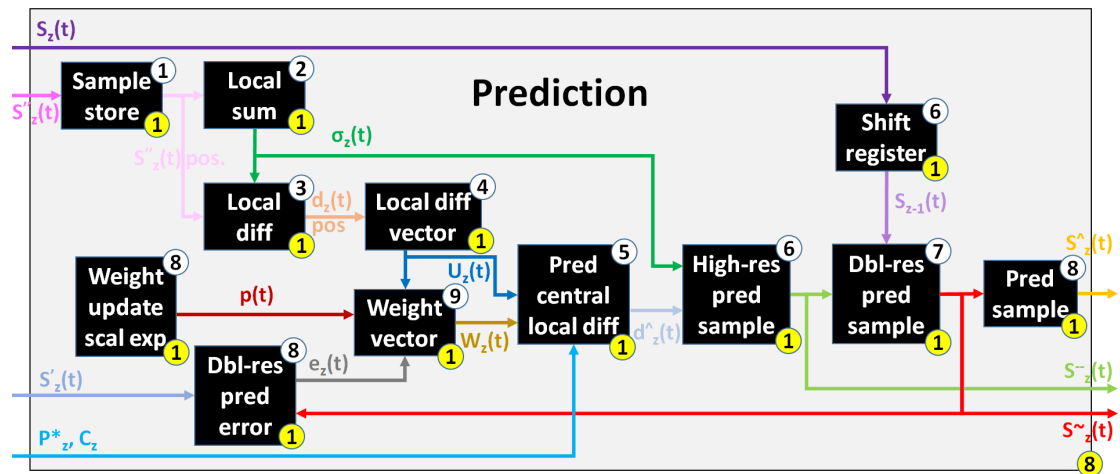


Figure 3.13: Prediction IP block diagram

Figure 3.13 shows the block diagram of the *Prediction IP*, by far the most complex one within the *Predictor* block, as it includes five times more IPs inside than any other sub-block.

Relatively speaking, this top IP simply interconnects all its sub-IPs, with no additional logic in between. Important to mention that, as one can see on Figure 3.13, these interconnections define the second (and last) close-loop branch in the whole algorithm, with all the dependencies, and again, this branch will not be executed until the open-loop branch finish processing.

The following operations are performed inside this IP, with the same execution order as here described:

1. Firstly, the *Sample store IP* bypasses the incoming *sample representative*  $s_z''(t)$  along with 5 specific surrounding (older) values at a given time (see section 3.7.5.1), to be used by the *Local sum IP* to compute a weighted *local sum*  $\sigma_z(t)$  out of them (see section 3.7.5.2).
2. The previous *local sum*  $\sigma_z(t)$ , together with the *surrounding sample representative*  $s_z''(t)$  values again, are taken to compute the (*central* and *directional*) *local differences*  $d_z(t)$  values (see section 3.7.5.3).
3. Next is creating the *local difference vector*  $U_z(t)$ , filled with the *local current* and *previous central local differences*  $P^*$  (see section 3.7.5.4).
4. The inner product of the *local differences*  $U_z(t)$  and *weights*  $W_z(t)$  vectors (this one coming from the close-loop, defined on the last step here below) produces the *predicted central local difference*  $\hat{d}_z(t)$  (see section 3.7.5.8), and along with the *local sum*  $\sigma_z(t)$  values one more time (delayed 3 clock cycles for synchro.), the *high-resolution predicted sample*  $\check{s}_z(t)$  value is generated too (see section 3.7.5.9).
5. Last but foremost, the *double-resolution predicted sample*  $\tilde{s}_z(t)$  value is computed based on the *original sample*  $s_z(t)$  and *high-resolution predicted sample*  $\check{s}_z(t)$  values (see section 3.7.5.10), to finally compute the real *predicted sample*  $\hat{s}_z(t)$  (see section 3.7.5.11).
6. Right at the end, executing now the close-loop branch, the *weights vector*  $W_z(t)$  is filled with the *weight*  $\omega_z$  values (one per *local difference*  $d_z(t)$ ), defining a starting value for each of them before updating them (see section 3.7.5.7). The updating process needs external values gathered from *Weight update scaling exponent IP* and *Double-resolution prediction error IP* (see sections 3.7.5.5 and 3.7.5.6, respectively).

All operations on the open loop branch define a total of 8 clock cycles to produce a valid output, given a valid input.

### 3.7.5.1 Samples Store IP

Figure 3.14 shows the block diagram of the *Sample Store IP*:

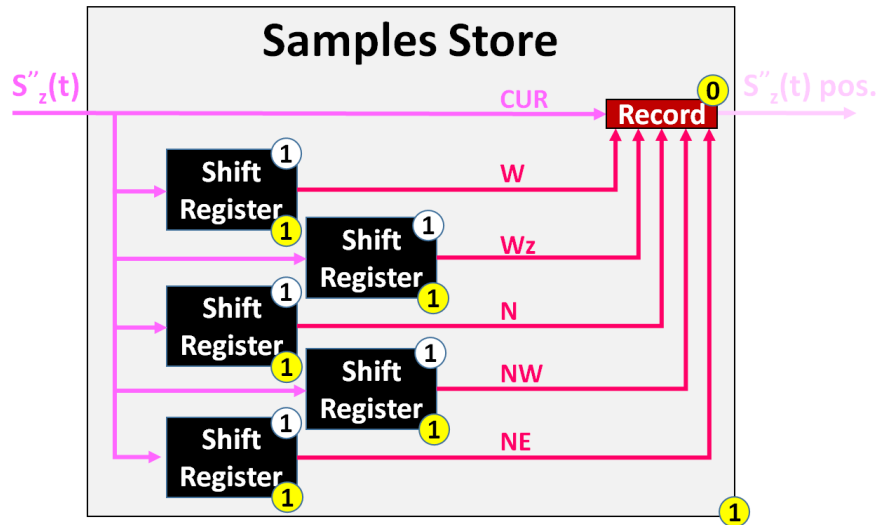


Figure 3.14: Samples Store IP block diagram

The *Samples Store IP* receives only a sample, the current *sample representative*  $s''_z(t)$  value in this case, and bypasses it again along with a total of 5 specific neighbour samples. In this context, neighbour samples imply that they are older samples, compared than the current one.

The neighbour samples are taken by instantiating the *Shift register IP* multiple times (one per neighbour sample) in parallel (see section 3.7.5.1.1), with the corresponding delay time configured at instantiation time for every one of them.

Compared to the incoming sample, with relative coordinates  $x=y=z=0$  (see Figure 2.1), these neighbour samples are:

- Position  $x=-1, y=0, z=0$ , abbreviated as 'W'.
- Position  $x=-1, y=0, z=-1$ , abbreviated as 'Wz'.
- Position  $x=0, y=-1, z=0$ , abbreviated as 'N'.
- Position  $x=-1, y=-1, z=0$ , abbreviated as 'NW'.
- Position  $x=+1, y=-1, z=0$ , abbreviated as 'NE'.

Of course, these locations are translated into delays that depend on the selected input sample order. Such values are detailed on Table 3.1:

Order	NW	N	NE	W	Z-1	Z-2	z-P
BSQ	$N_X+1$	$N_X$	$N_X-1$	1	$N_X*N_Y$	$2*N_X*N_Y$	$P*N_X*N_Y$
BIP	$(N_X+1)*N_Z$	$N_X*N_Z$	$(N_X-1)*N_Z$	$N_Z$	1	2	P
BIL	$N_X*N_Z+1$	$N_X*N_Z$	$N_X*N_Z-1$	1	$N_X$	$2*N_X$	$P*N_X$

Table 3.1: Delay values to generate all neighbour samples

The 5 neighbour samples together with the current sample are put within a record (combinational logic) before being sent out.

This IP just needs 1 clock cycle to produce a valid output, given a valid input, but there is one important idea to keep in mind, addressed in section 3.7.5.1.1.

### 3.7.5.1.1 Shift Register IP

The *Shift Register IP*, shown in Figure 3.14, is a clocked process that basically introduces the incoming sample value in the least significant position of an array, whose size is as long as desired (see Table 3.1 for the meaningful values), and it moves such value to the next/left position of the array on every clock cycle. The output of the module is the most significant position of such array, as Figure 3.15 shows:

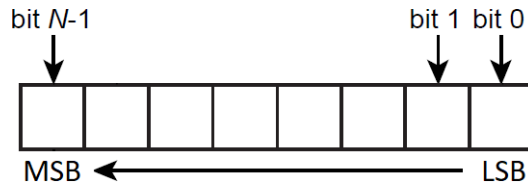


Figure 3.15: Shift register structure

The size of the sample, its default value and the size of the array (equivalent to the delay time in clock cycles) are freely configurable by the user at instantiation time.

As a result, with this IP it is possible to get an older specific sample at a given moment in time, or many of them if instantiations in parallel are performed instead (see section 3.7.5.1). The *enable* and *image coordinates* are delayed the same quantity of time too, so that next IPs in the chain have the possibility to start working once the configured delay is reached.

While it is true that this IP only needs 1 clock cycle to provide a new output value, it must be noted that such output will always be 0 until the array is fully filled for the very first time. In other words, the delay time (in clock cycles) is reached.

Nevertheless, this is something not to worry about, as thanks to the *Image Coordinates Control IP*, the algorithm itself is smart enough to know when the neighbour samples are whether available or not. Equation 2.25 is a good example of this.

Section 10.14 shows part of its source code.

### 3.7.5.2 Local Sum IP

Figure 3.16 shows the block diagram of the *Local Sum IP*:

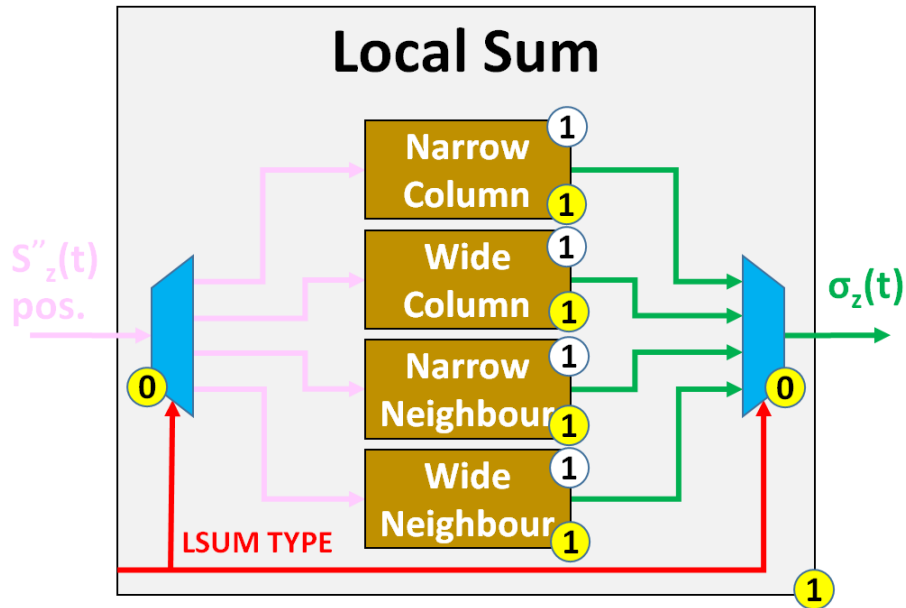


Figure 3.16: Local Sum IP block diagram

This IP computes the weighted *local sum*  $\sigma_z(t)$  value out from some *neighbour sample representative*  $s_z''(t)$  values in *spectral band*  $z$ , depending on the selected configuration. Neighbour positions are North (N), West (W) and North-West (NW).

The user defines how to compute this sum, and then the IP only instantiates the selected clocked-process, as this option is not allowed to change at run-time:

1. *Wide neighbor-oriented* local sum option (see equation 2.24).
2. *Narrow neighbor-oriented* local sum option (see equation 2.25).
3. *Wide column-oriented* local sum option (see equation 2.26).
4. *Narrow column-oriented* local sum option (see equation 2.27).

This unique instantiated clocked process defines that just 1 clock cycle is required to produce a valid output, given a valid input.

Finally, in order to avoid confusions with the user configuration, the constraints defined on section 2.3.5.1 are not implemented by this IP itself, but instead they are implemented on the very top entity (see section 3.5).

This is done to ensure a unique absolute configuration in the whole system, not changing depending on the sub-IP that the developer is checking at that very moment.

### 3.7.5.3 Local Differences IP

Figure 3.17 shows the block diagram of the *Local Differences IP*:

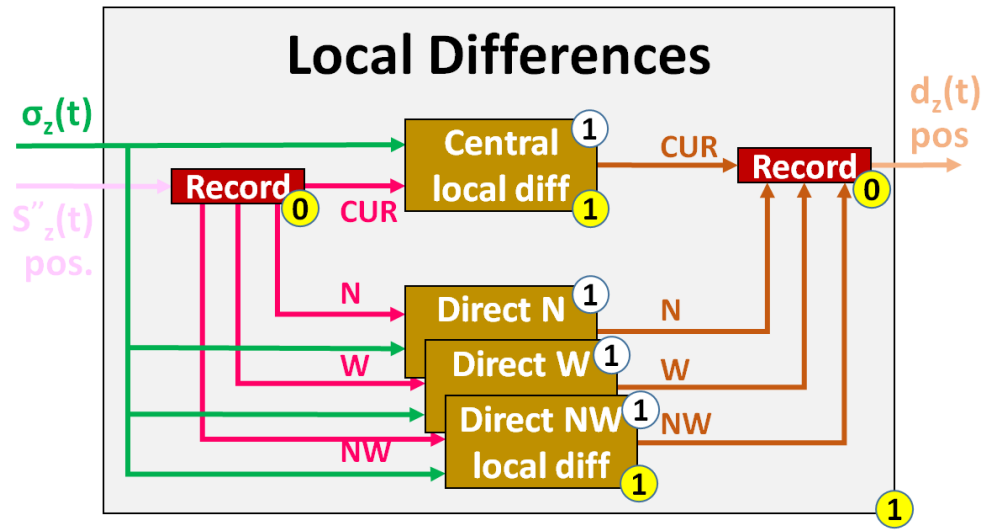


Figure 3.17: Local Differences IP block diagram

This IP computes the *local difference*  $d_z(t)$  values by means of the *local sum*  $\sigma_z(t)$  and *sample representative*  $s_z''(t)$  values.

The *local difference*  $d_z(t)$  and *sample representative*  $s_z''(t)$  values are managed by their respective custom record types, which store together the *central* (or *current*) and *directional* (neighbour) sample values of them. Neighbour positions are North (N), West (W) and North-West (NW).

As one can see on Figure 3.17, there are four different clocked processes, working all of them in parallel, to compute the *central local difference* (see equation 2.28) as well as the three *directional local differences*  $d_z^N(t)$ ,  $d_z^W(t)$  and  $d_z^{NW}(t)$  (see equations 2.29, 2.30 and 2.31, respectively).

If the user configures the system to work under *Reduced Prediction mode*, *directional local differences*  $d_z^N(t)$ ,  $d_z^W(t)$  and  $d_z^{NW}(t)$  are permanently fixed to 0.

The two records (pure combinational logic) are used to extract the sub-signals from the *sample representative*  $s_z''(t)$  array before the computation, and to join the sub-signals from the *local difference*  $d_z(t)$  array after the computation.

Because of all the processes are executed in parallel, and the other parts of the IP are combinational logic, only 1 clock cycle is required in to produce a valid output, given a valid input.



### 3.7.5.4 Local Differences Vector IP

Figure 3.18 shows the block diagram of the *Local Differences Vector IP*:

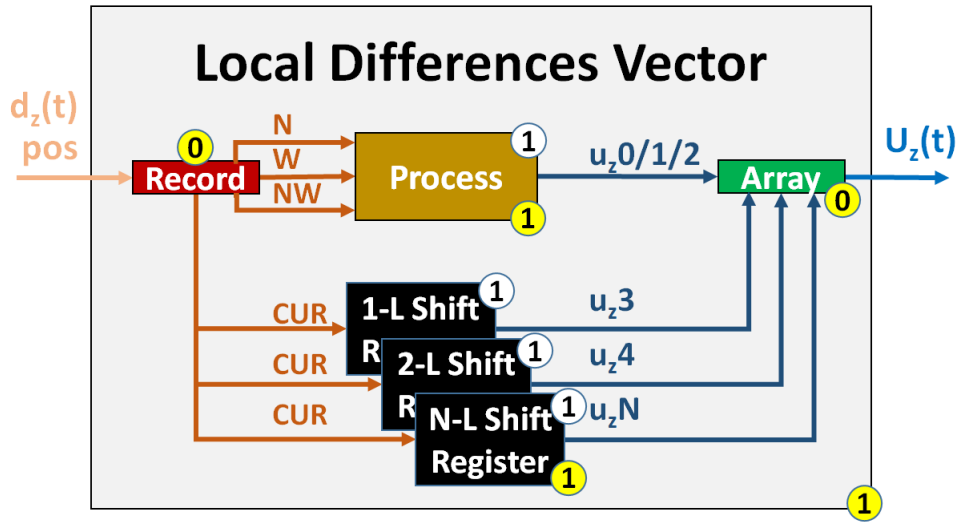


Figure 3.18: Local Differences Vector IP block diagram

This IP takes the *central* and *directional local difference*  $d_z(t)$  values and it computes the *local difference vector*  $U_z(t)$ , as described on section 2.3.5.3.

Regardless of choosing either *Reduced* or *Full prediction mode*, as well as the number of *previous spectral bands*  $P^*$  to use (see equation 2.32), the vector  $U_z(t)$  is directly created for the longest possible case (full prediction mode and  $P = 15$ ).

This approach makes this module quite simpler and still efficient, and later, the *Predicted Central Local Difference IP* will fetch only of the necessary array positions, depending on the *current spectral band*  $z$  (see section 3.7.5.8).

Managed by a clocked process, the incoming *directional local differences*  $d_z^N(t)$ ,  $d_z^W(t)$  and  $d_z^{NW}(t)$  are placed into the 3 first positions of the *local difference vector*  $U_z(t)$ , or simply set to 0 if working under *Reduced prediction mode*.

To compute all previous *central local difference*  $d_z(t)$  values, 15 (longest case) *Shift register IP*s are instantiated, setting each one with delay  $z-1$  (see Table 3.1) and being its output the input of the next one.

In the same way as with the *Local Differences IP*, the input and output data arrays extract and join their sub-signals, before and after computation, respectively.

Similar to the *Samples Store IP* (see section 3.7.5.1), the *Shift Register IP*s here used to generate the *previous central local differences* will require some additional clock cycles before the proper values are outputted. Nevertheless, the algorithm controls the *image coordinates* and knows when the required data is available.

The clocked process and sub-IPs are all executed in parallel, and so, only 1 clock cycle is required to produce a valid output, given a valid input.

### 3.7.5.5 Weight Update Scaling Exponent IP

The *Weight Update Scaling Exponent IP*, placed in the close-loop branch from Figure 3.13, implements a clocked process to compute the *weight update scaling exponent*  $p(t)$  value, as equation 2.47 defines.

Unlike the rest of the IPs, the output value does not depend on other signals, but entirely on the *image coordinates* and user-specified parameters  $v_{min}$ ,  $v_{max}$  and  $t_{inc}$ , which are constrained according to equations 2.48 and 2.49.

This calculation is broken down into smaller steps for the sake of readability and debugging purposes.

This simply IP just needs 1 clock cycle to produce a valid output, given a valid input.

### 3.7.5.6 Double-Resolution Prediction Error IP

The *Double-Resolution Prediction Error IP*, placed in the close-loop branch from Figure 3.13, implements a clocked process to compute the *double-resolution prediction error*  $e_z(t)$  value, as equation 2.50 states.

This calculation is broken down into smaller steps for the sake of readability and debugging purposes.

This simply IP just needs 1 clock cycle to produce a valid output, given a valid input.

### 3.7.5.7 Weights Vector IP

Figure 3.19 shows the block diagram of the *Weights Vector IP*:

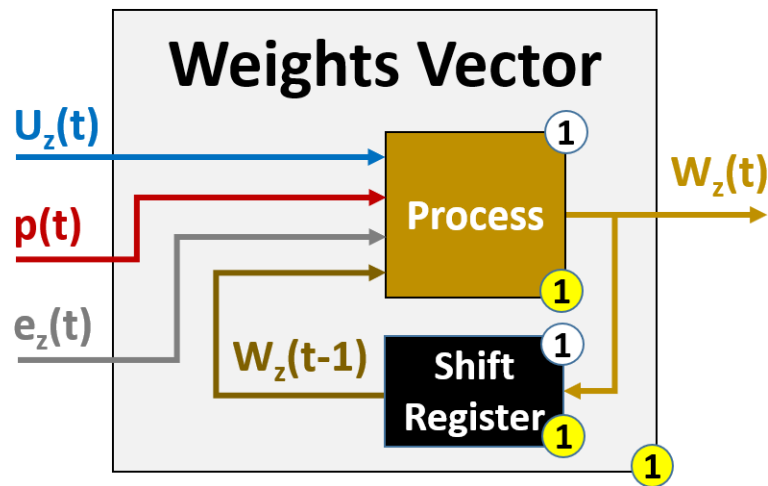


Figure 3.19: Weights Vector IP block diagram

This IP implements a clocked process for the initialization and the update of the *weight*  $\omega_z$  values, to be placed within the *Weights vector*  $W_z(t)$ , as described on sections 2.3.5.4 and 2.3.5.7.

As defined on section 2.3.5.4, the configured *weights initialization* type, *default* or *custom*, is applied at the beginning of all *spectral bands*  $z$  (every time that  $t = 0$ ), computing equations 2.37 and 2.38 for the *default initialization* (for either *full* or *reduced prediction mode*), and equation 2.39 for the *custom initialization*. In any case, the resulting values are constrained as equation 2.36 states.

Right away after initialization, the *weight*  $\omega_z$  values must be updated on each clock cycle, according to equations 2.41 (*central weight value*), 2.42, 2.43 and 2.44 (*directional weight values*). These equations are broken down in smaller steps with variables to make the computations easy to follow and debug.

As these equations require the previous *weight*  $\omega_z$  values on  $t - 1$  to compute the new ones, a *Shift Register IP* (see section 3.7.5.1.1) is instantiated, and configured depending on the samples input order, to delay the signal properly (see Table 3.1).

This updating process demands the *weight update scaling exponent*  $p(t)$  and *double-resolution prediction error*  $e_z(t)$  values, both of them computed one clock cycle before (see sections 3.7.5.5 and 3.7.5.6, respectively).

Finally, all *weight*  $\omega_z$  values are placed within the *weight vector*  $W_z(t)$ . The *directional weight*  $\omega_z^N(t)$ ,  $\omega_z^W(t)$  and  $\omega_z^{NW}(t)$  values are placed into the 3 first positions from the vector, and the other positions are filled with the *central weight*  $\omega_z(t)$  values (one per *spectral band*  $z$ ). This *weight vector*  $W_z(t)$  actually has the same positions as the *local difference vector*  $U_z(t)$ .

Indeed, following the same approach as for the *Local Difference Vector* IP, this vector is directly created for the longest possible case (*full prediction mode* and  $P = 15$ ), regardless of the chosen *prediction mode* and *previous spectral band*  $P^*$  to use (see equation 2.32).

Later, the *Predicted Central Local Difference IP* (see section 3.7.5.8) will fetch only the necessary positions, depending on the current *spectral band*  $z$ .

With the clocked process and the sub-IP, both of them executed in parallel, the *Weights Vector IP* just requires 1 clock cycle to produce a valid output, given a valid input.

### 3.7.5.8 Predicted Central Local Difference IP

Figure 3.20 is the block diagram of the *Predicted Central Local Difference IP*.

This IP is a clocked process that computes the *predicted central local difference*  $\hat{d}_z(t)$  value, which is the inner product between the *local difference vector*  $U_z(t)$  and the *weight vector*  $W_z(t)$ , producing a single output value per both complete input vectors, just as equation 2.53 depicts.

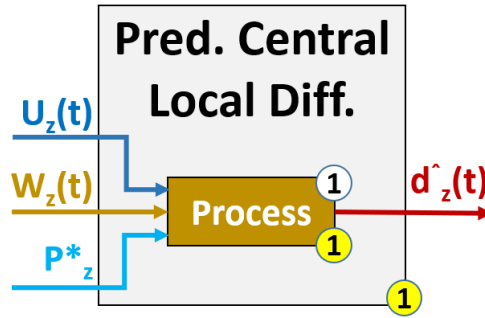


Figure 3.20: Predicted Central Local Difference IP block diagram

As already mentioned on sections 3.7.5.4 and 3.7.5.7, the *Local Differences Vector IP* and *Weights Vector IP* are generated for the longest case ( $P = 15$ ) regardless of the configuration, but the *Predicted Central Local Difference IP* must take into account only the necessary positions to compute equation 2.53 properly.

For such a purpose, the IP uses the *preceding spectral bands*  $P_z^*$  value and the *image coordinates*. The inner product consists of a for-loop that multiplies the same position of both vectors, and adds the result to the next iteration multiplication. Therefore, the for-loop size is basically the current number of *spectral band*  $z$  (being *preceding spectral bands*  $P_z^*$  the maximum value).

The selected *prediction mode* is read at instantiation time, and so, if the current *spectral band*  $z$  is the first one ( $z = 0$ ) while working under *Reduced prediction mode*, the output is fixed to  $\hat{d}_z(t) = 0$ .

This IP just needs 1 clock cycle to generate a valid output, given a valid input.

### 3.7.5.9 High-Resolution Predicted Sample IP

The *High-Resolution Predicted Sample IP*, shown in Figure 3.13, computes the *high-resolution predicted sample*  $\check{s}_z(t)$  value, as equation 2.54 defines. The computation is broken down into smaller steps for simplicity and debugging purposes.

This IP just needs 1 clock cycle to generate a valid output, given a valid input.

### 3.7.5.10 Double-Resolution Predicted Sample IP

The *Double-Resolution Predicted Sample IP*, shown in Figure 3.13, computes the *double-resolution predicted sample*  $\tilde{s}_z(t)$  value, as equation 2.56 shows.

This IP just needs 1 clock cycle to generate a valid output, given a valid input.

### 3.7.5.11 Predicted Sample IP

The *Predicted Sample IP*, shown in Figure 3.13, computes the *predicted sample*  $\hat{s}_z(t)$  value, according to the equation 2.57.

This IP just needs 1 clock cycle to generate a valid output, given a valid input.

### 3.8 Predictor-Encoder Interconnection IP

Figure 3.21 shows the block diagram of the *Predictor-Encoder Interconnect IP*:

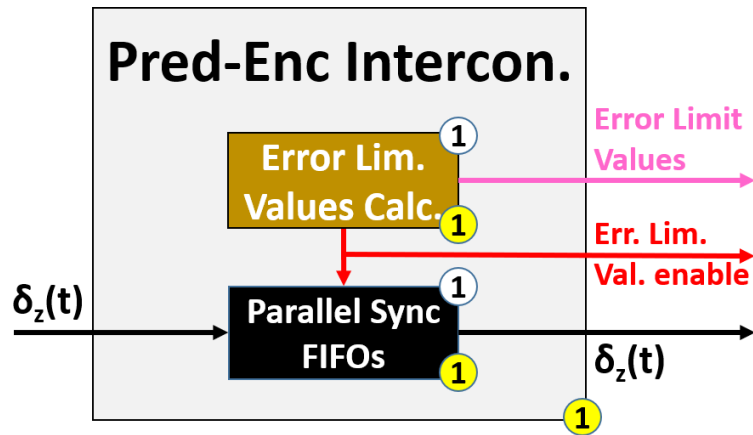


Figure 3.21: Predictor-Encoder Interconnect IP block diagram

As its identification name suggests, this IP is used as a bridge between the *Predictor* and *Encoder* blocks. It is required only when the *Periodic Error Limit Updating* option is enabled by the user configuration (see section 2.3.2.1.1), as the *Encoder* block would expect not only the *mapped quantizer index*  $\delta_z(t)$  values from the *Predictor* block, but the *Error Limit Values* as well.

If such option is not enabled, the *Predictor* and *Encoder* blocks are connected straightforward (single connection for the *mapped quantizer index*  $\delta_z(t)$  values).

This IP is in charge of two operations, both executed in parallel:

1. A clocked process updates and outputs the *Error Limit Values* according to the user configuration (*absolute*, *relative* or both error limit types) every  $U$  frames (see section 2.3.2.1.1).
  - While the *Error Limit Values* are being outputted, an *enable* signal is also sent out to the *Encoder* block as well as to the *Parallel Synchronous FIFOs* IP, so that the next IPs in the chain are aware of the kind of received data.
2. The *Parallel Synchronous FIFOs* IP continuously stores and releases the incoming *mapped quantizer index*  $\delta_z(t)$  values and *image coordinates*, except when it is time to output the *Error Limit Values*, moment at which such data stall until its next turn (see section 3.8.1).

As the clocked process and the sub-IP are executed both in parallel, the *Predictor-Encoder Interconnection IP* just needs 1 clock cycle to generate a valid output, given a valid input.

### 3.8.1 Parallel Synchronous FIFOs IP

Figure 3.22 shows the block diagram of the *Parallel Synchronous FIFOs IP*:

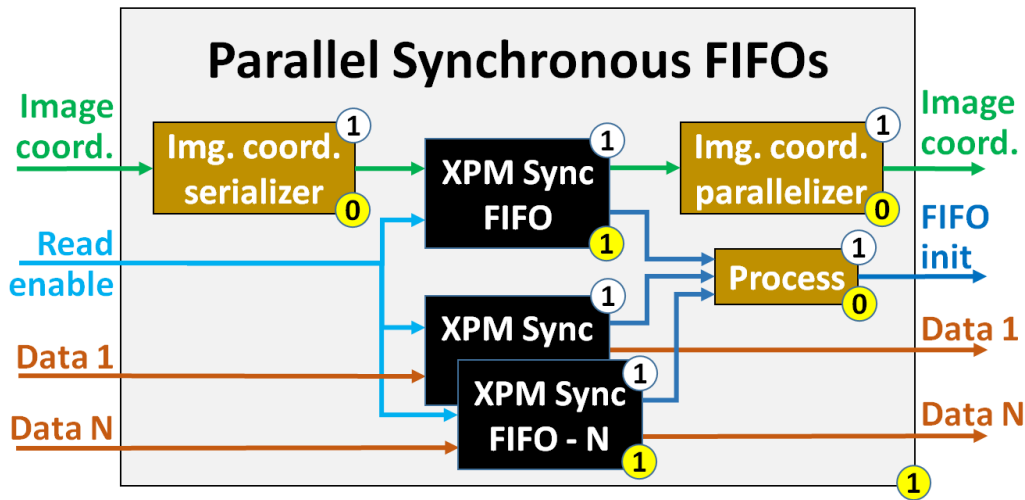


Figure 3.22: Parallel Synchronous FIFOs IP block diagram

This IP is the only one integrating external sub-IPs, the *Synchronous FIFO* from the XPM library v2019.1 [32]. It is in charge of storing different kind of data (one per *XPM FIFO*), but sorted out just in two types:

1. Image coordinates: This first *XPM FIFO* is prepared to store exclusively the incoming *image coordinates*, to control the position of the other data stored.
2. General data: A configurable number of *XPM FIFOs* are instantiated in parallel to store any other kind of data.

As these FIFOs can accept data only in *std\_logic\_vector*, such format should previously be given, if required. This is not needed for the *General data* because it is already in *std\_logic\_vector*, but for the *image coordinates* it is different.

The *image coordinates* are stored in an array with the x, y, z and t positions (see section 3.6), so they need 2 non-clocked processes: one to serialize such positions (before entering the FIFO) and another one to parallelize them (after leaving the FIFO).

The FIFOs must always be ready to write data in, but this data must be read out only under certain circumstances (see sections 3.8 and 3.9.3), so a *Read enable* input signal is integrated for such a purpose.

Besides, as these FIFOs need some time to initialize [32, p.47], an additional *AND gate* merges their *rst\_busy* signals into *FIFO init* output signal, to inform to the rest of the design when the *Parallel Synchronous FIFOs IP* is ready to start working.

All FIFOs are executed in parallel and the rest of logic is combinational, so this IP (once FIFOs were initialized) just need 1 clock cycle to produce a valid output, given a valid input. Section 10.16 shows part of its source code.

### 3.9 Encoder Top IP

The *Encoder IP* implements the encoding stage of the compressor and the format of such output image, described on section 2.4. Figure 3.23 shows its block diagram:

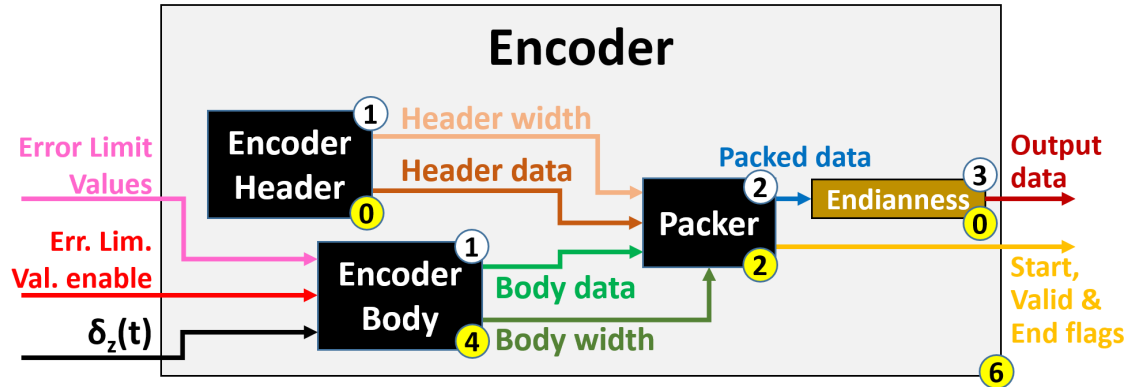


Figure 3.23: Encoder Top IP block diagram

The *Encoder IP* functionality is broken down into the following tasks:

1. First of all, the *Encoder Header IP*, which is purely combinational (see section 3.9.1), generates the variable-length header according to the user configuration.
2. The *Encoder Body IP* (see section 3.9.2) encodes the incoming *mapped quantizer index*  $\delta_z(t)$  values, sample by sample.
  - Additionally, if the *Periodic Error Limit Updating* option is enabled, then the *Error Limit Values* are periodically encoded as part of the body as well.
3. Next, the *Packer IP* (see section 3.9.3) fetches both the encoder header and body data (plus their respective widths) to pack them with the user-specified width  $B$ .
4. Finally, the output packets go into a non-clocked (combinational) process for the Endianness, to either do nothing (bytes already on Big Endian order) or rearrange the bytes to Little Endian order.

While it is true that the *Encoder Header IP* and *Encoder Body IP* start both at the very beginning in parallel, the *Encoder Header IP* produces an output immediately (as it has no input signals, just a static configuration), but the *Encoder Body IP* must wait until the *Predictor IP* starts to send data to it.

Finally, in order to make the system more configurable according to the user needs, a new user-specified parameter has been added to whether instantiate the *Encoder Header IP* or not. This option allows to simplify the design and to work only with the desired part of the *Encoder* block.

The critical path of this IP (with *Hybrid Entropy Coder* selected) defines a total of 6 clock cycles to generate a valid output, given a valid input.

### 3.9.1 Encoder Header IP

Figure 3.24 shows the block diagram of the *Encoder Header IP*:

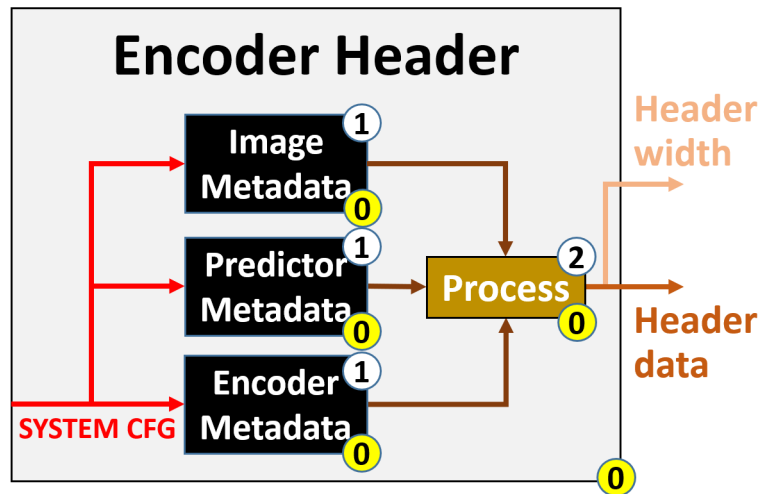


Figure 3.24: Encoder Header IP block diagram

This IP instantiates the 3 *Metadata* sub-IPs (*Image*, *Predictor* and *Encoder*) altogether with the aim to fetch and output all header data together. The whole design is purely combinational logic (not even clock and reset signals exist here) and it just holds static configuration, so the payload is already arranged from the very beginning.

It must be noted that the 3 IPs output their data with a variable size, depending on the configuration, something that enters in conflict when declaring the VHDL entity. To overcome this problem of entity declaration, each IP must output both the payload itself (in a very-long fixed-size signal) and, in parallel, there is another signal informing about the meaningful data-bits from the payload signal.

This is where the non-clocked process from the end of the chain comes into play. It takes the *Image*, *Predictor* and *Encoder Metadata*s, together with their meaningful data size, and it serializes and outputs them by just joining the meaningful data-bits.

But once more, the resulting serialized payload is introduced and outputted in a very-long fixed-size signal (4096-bits for security), and the sum of the 3 metadata sizes outputted as integer in parallel as well.

Additionally, to avoid strange values, both outputs are set to 0 until the sub-IPs have meaningful data to offer.

Because of the nature of the *Encoder Header IP*, 0 clock cycles (so immediately) are necessary to generate a valid output.



### 3.9.1.1 Image Metadata IP

The *Metadata IPs* are in charge of implementing Tables 2.1 to 2.16 and serialize them into a single output signal, together with another signal in parallel that simply defines the size of the mentioned payload.

To make this possible (logic applicable to the rest of *Metadata IPs*), the source code of each one is structured as follows:

1. A custom record type is created for every table, with the same data fields as the table sub-elements, plus another one with the total data-bits width of it.
  - If a table has a variable-length field (like *Table Data Subblock* field within Table 2.5), such custom record type will have a very-long fixed-size signal for this payload, plus an extra field to store its meaningful data-bits size.
  - If a table has another table inside (like Table 2.6 within Table 2.9), the custom record types are nested as well.
2. Constants of these custom record types are created, and initialized with the necessary data. Then, each constant represents a table.
  - For the initialization of every variable-length field, two functions are implemented: one to generate the payload itself, and another one to calculate its total size.
3. All constants are serialized, each one of them with its additional dedicated function, together into the output signal, and the previous total data-bits width field is used to inform about its meaningful data size too.

Thus, for the specific case of the *Image Metadata IP* (in Figure 3.24), Tables 2.2, 2.3 and 2.5 (see sections 2.4.1.1 and 2.4.1.1.1) are instantiated, filled, and serialized into an output signal (whose size is 2048-bits to ensure all possible configurations fit), together with another signal that defines the payload size.

All necessary data are static configuration values coming from different VHDL packages, including the *Supplementary Information Tables* integrated here inside as well (see section 3.9.1.1.1).

This IP is purely combinational, so this header data is generated immediately. Section 10.15 shows part of its source code.

### 3.9.1.1.1 Supplementary Information Tables

According to section 2.4.1.1.1, the CCSDS-123 standard foresees the possibility to define up to 15 variable-length tables with auxiliary information to an end user, such as the wavelength associated with each *spectral band*  $z$  [10, p.22-24].

When enabled, each table demands a preconfigured *purpose* (see Table 2.4), *structure* (0-dimensional, 1-dimensional, 2-dimensional ZX or 2-dimensional YX) and *data type* (unsigned, signed or float).

All these requirements are static configuration that the user must define before executing the algorithm. Thus, and in the same way as in section 3.7.2.1.1, a VHDL package is created covering all the possibilities:

- 15 constant 0-dimensional arrays of unsigned data type.
- 15 constant 0-dimensional arrays of signed data type.
- 15 constant 0-dimensional arrays of float data type.
- 15 constant 1-dimensional arrays of unsigned data type.
- 15 constant 1-dimensional arrays of signed data type.
- 15 constant 1-dimensional arrays of float data type.
- 15 constant 2-dimensional ZX arrays of unsigned data type.
- 15 constant 2-dimensional ZX arrays of signed data type.
- 15 constant 2-dimensional ZX arrays of float data type.
- 15 constant 2-dimensional YX arrays of unsigned data type.
- 15 constant 2-dimensional YX arrays of signed data type.
- 15 constant 2-dimensional YX arrays of float data type.

Each one of these arrays are placed into an independent matrix, so that there is one single place where to extract the 15 arrays of a group, and the user configuration defines which matrix is the one to be used.

As a VHDL package, the *Supplementary Information Tables* are purely combinational logic, so this header data is available immediately.

### 3.9.1.2 Predictor Metadata IP

Using exactly the same structure as detailed on section 3.9.1.1, the *Predictor Metadata IP*, shown in Figure 3.24, is in charge of instantiating Tables 2.6 to 2.14 (see section 2.4.1.2), filled them with the proper data and serialize them in the same order as listed within such tables.

There are 2 specific functions for every existing variable-length field: one for generating the payload, and another one for computing its data size. In this case, the variable-length fields are: *Absolute Error Limit Values Subblock*, *Relative Error Limit Values Subblock*, *Damping Table Subblock* and *Offset Table Subblock* fields.

Even though all existing tables are instantiated and filled here, not all of them are included into the output signal, but just the enabled ones, and most of the tables have conditional fields. User configuration parameters *Weight Initialization Table Flag*, *Weight Exponent Offset Table Flag*, *Damping Table Flag* and *Offset Table Flag* are the conditional factors for such purpose.

In the same way as with the *Image Metadata IP*, the outputs of this IP are the complete serialized header payload on a very-long fixed-size signal (2048-bits because of all available tables) in parallel with another signal that says the meaningful data-bits from payload.

This IP is purely combinational logic, so this header data is generated immediately.

### 3.9.1.3 Encoder Metadata IP

Using exactly the same structure as detailed on section 3.9.1.1, the *Encoder Metadata IP*, shown in Figure 3.24, is in charge of instantiating Tables 2.15 and 2.16 (see section 2.4.1.3), filled them with the proper data and serialize them in the same order as listed within such tables.

There are 2 specific functions for every existing variable-length field: one for generating the payload, and another one for computing its data size. Here, the variable-length field is: *Accumulator Initialization Table* field from *Sample-Adaptive Entropy Coder* table.

There is one extra function here to give the same format to the *Encoder Metadata IP* regardless of the selected *Entropy Coder* (as each table is slightly different), so that the *Encoder Header IP* can process it with the same logic.

Both tables are instantiated from the very beginning, but only either Table 2.15 or 2.16 is included into the output signal, depending on *Sample-Adaptive Entropy Coder* or *Hybrid Entropy Coder* has been selected, respectively.

In the same way as with the *Image Metadata IP*, the outputs of this IP are the serialized header payload on a very-long fixed-size signal (512-bits because this encoder is quite short) in parallel with another signal that says the meaningful data-bits of the payload.

This IP is purely combinational logic, so this header data is generated immediately.

### 3.9.2 Encoder Body IP

Figure 3.25 shows the block diagram of the *Encoder Body IP*:

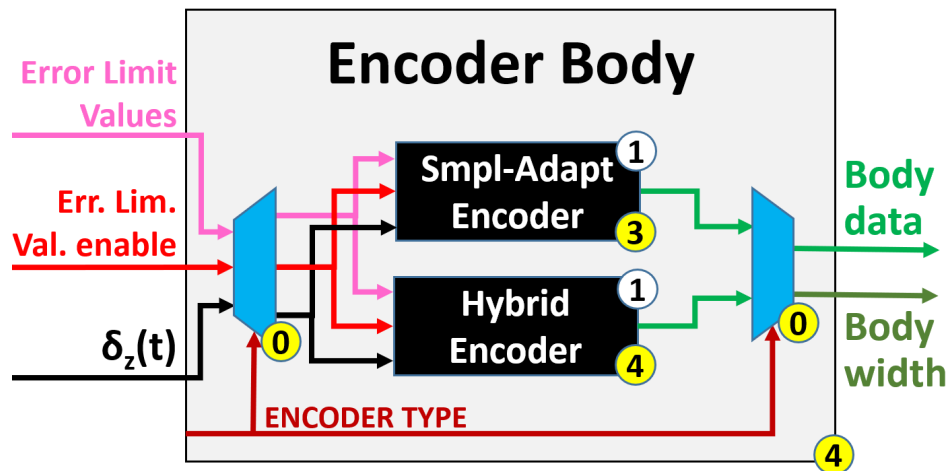


Figure 3.25: Encoder Body IP block diagram

This IP instantiates one of the available entropy coders, and it forwards both the input and output signals, with no additional logic at all in between.

In other words, it just acts as a wrapper for the selected sub-IP.

The available entropy coders are <sup>2</sup>:

- *Sample-Adaptive Entropy Coder IP*, whose implementation is explained on section 3.9.2.1.
- *Hybrid Entropy Coder IP*, whose implementation is explained on section 3.9.2.2.

In order to forward the input and output signals, a multiplexer and a demultiplexer are used (pure combinational logic), routing to/from the selected entropy coder.

As a safety measurement, in case no entropy coder has been configured, this IP sets the output signals to 0, and *enable* signal and *image coordinates* are not forwarded either. This is done to ensure that no strange behaviour is experienced on the next IPs in the chain if a bad configuration was provided.

The *Encoder Body IP* totally depends on the selected *Entropy Coder* to define the necessary number of clock cycles to produce a valid output, given a valid input. Therefore, depending on the user configuration, refer to section either 3.9.2.1 or 3.9.2.2 to see the timing characteristics of this IP.

<sup>2</sup>*Block-Adaptive Entropy Coder* is not mentioned here, because it was not implemented.

### 3.9.2.1 Sample-Adaptive Entropy Coder IP

Figure 3.26 shows the block diagram of the *Sample-Adaptive Entropy Coder IP*:

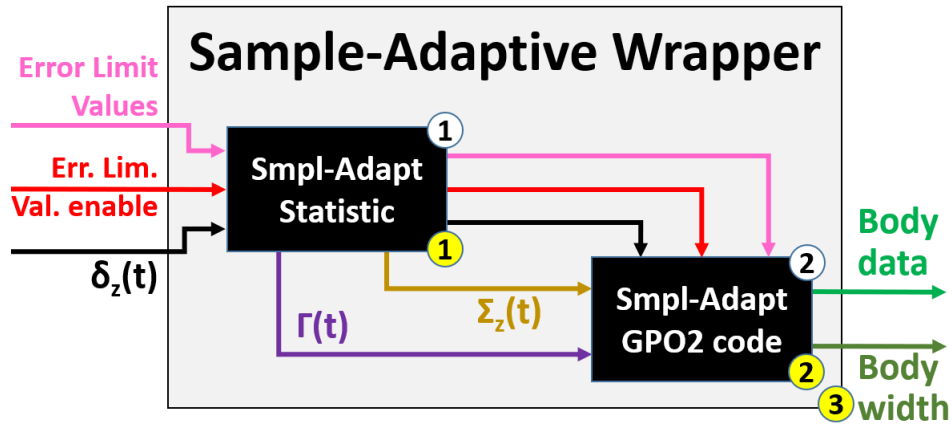


Figure 3.26: Sample-Adaptive Entropy Coder IP block diagram

This IP interconnects its sub-IPs with no additional logic in between, and it is in charge of encoding the *mapped quantizer index*  $\delta_z(t)$  and *Error Limit values* (both referred as payload) under the *Sample-Adaptive Entropy Coder* logic.

The following operations are performed, with the same execution order as listed here:

1. First, the *Sample-Adaptive Statistic IP* uses the *mapped quantizer index*  $\delta_z(t)$  values to compute the *accumulator*  $\Sigma_z(t)$  and *counter*  $\Gamma(t)$  values (see section 3.9.2.1.1).
2. Second, the *Sample-Adaptive GPO2 Coder IP* uses *mapped quantizer index*  $\delta_z(t)$  and *Error Limit values* (delayed one clock cycle to be synchronized) along with the *accumulator*  $\Sigma_z(t)$  and *counter*  $\Gamma(t)$  values to output the payload *codeword*  $R_{k_z(t)}(\delta_z(t))$  as well as its meaningful data size (see section 3.9.2.1.2).

The *Sample-Adaptive Entropy Coder IP* needs a total of 3 clock cycles to produce a valid output, given a valid input.

#### 3.9.2.1.1 Sample-Adaptive Statistic IP

Figure 3.27 shows the block diagram of the *Sample-Adaptive Statistic IP*. This IP uses the *mapped quantizer index*  $\delta_z(t)$  values on a clocked process to compute the *accumulator*  $\Sigma_z(t)$  and *counter*  $\Gamma(t)$  values, according to equations 2.71, 2.72, 2.74 (for initialization), 2.75 and 2.76 (for updating).

It can be seen in these equations from above that data on position  $t - 1$  is necessary to compute the next values. But as such previous position on time depends on the selected samples input order (see Table 3.1), 5 *Shift Register IPs* in parallel (one per input and output signal) are instantiated and configured with the (same) right delay.

One can appreciate that *Error limit values* and its *enable* signal are not required in this IP, so they are simply delayed and outputted for the next IP in the chain. As this delay depends on the selected samples input order, they use *Shift Register IPs* with the same configuration for such operation.

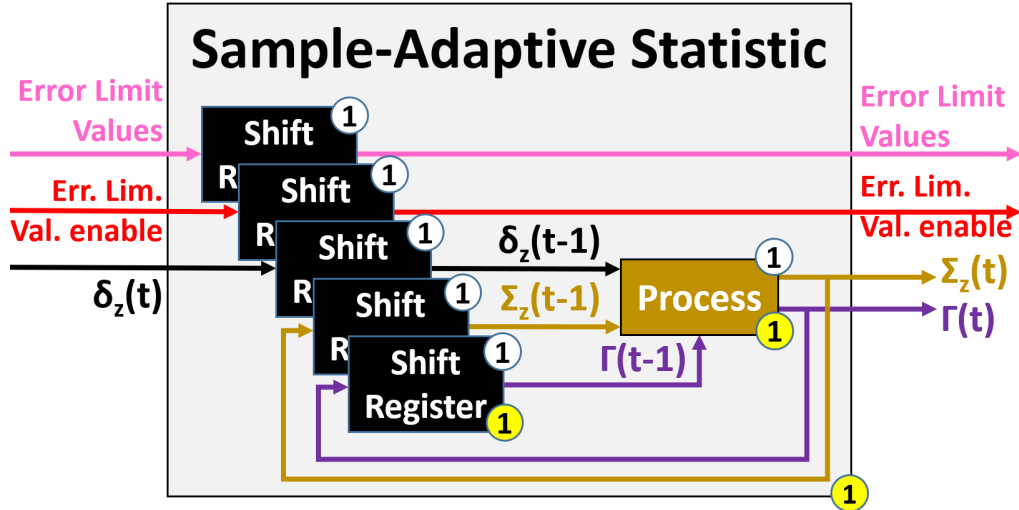


Figure 3.27: Hybrid Entropy Coder IP block diagram

This IP just needs 1 clock cycle to generate a valid output, given a valid input. Nonetheless, one more time it is important to keep in mind that this IP, as any other one using the *Shift Register IP*, outputs no data until the configured delay time is reached, but the algorithm is smart enough to know when to demand the data (see section 3.7.5.1.1).

### 3.9.2.1.2 Sample-Adaptive GPO2 Coder IP

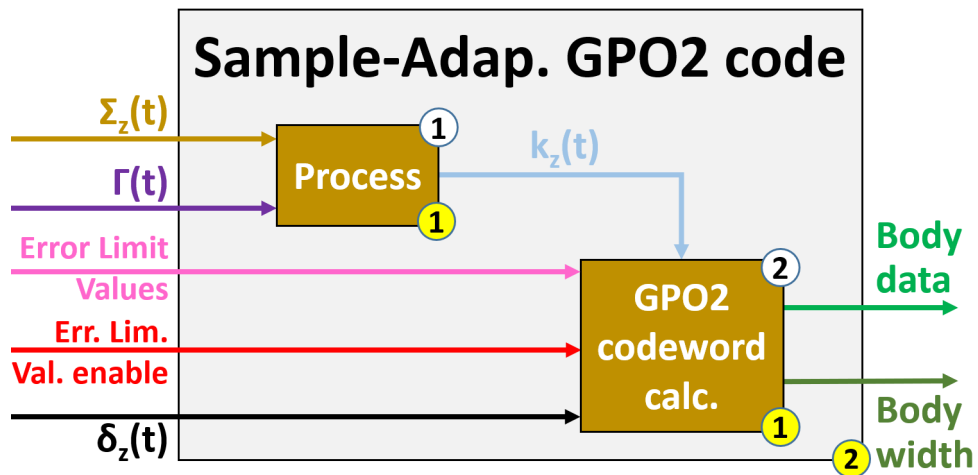


Figure 3.28: Sample-Adaptive GPO2 Code IP block diagram

Figure 3.28 shows the block diagram of the *Sample-Adaptive GPO2 Code IP*. This IP uses the incoming payload (*mapped quantizer index*  $\delta_z(t)$  and *Error limit values*) to generate the variable-length *GPO2 codeword*  $R_{k_z(t)}(\delta_z(t))$ .

There are a total of two clocked processes for four different operations, executed in the same order as listed here:

1. The first clocked process uses the incoming *accumulator*  $\Sigma_z(t)$  and *counter*  $\Gamma(t)$  values to compute the *variable-length code parameter*  $k_z(t)$ , operation defined on equation 2.79, and result constrained as equation 2.80 shows.
2. The second clocked process is in charge of three operations, managed by the incoming *Error Limit values enable* signal:
  - a) If the *Error Limit values enable* signal is high (just one clock cycle every  $U$  frames, see section 2.4.2), new *absolute* and/or *relative Error Limit values* are received, delayed one clock cycle to synchronize, and encoded with  $D_A$  and  $D_R$  bits, respectively.
    - The quantity of *Error limit values* to encode vary from one single value to a couple of arrays of size  $N_Z$  each, as section 3.7.2.1.1 explains.
  - b) If the *Error Limit values enable* signal is low, then the *GPO2 codewords*  $R_{k_z(t)}(\delta_z(t))$  are computed by using the *mapped quantizer index*  $\delta_z(t)$  values, delayed one clock cycle to synchronize, together with the *variable-length code parameter*  $k_z(t)$ , as described on section 2.4.2.1.2.
    - As this codeword has a variable-length, its meaningful data-bits size must be outputted in parallel with it.
  - c) At the very end, once the complete image has been encoded, both output signals are fixed to 0. This small tweak makes the *Packer IP* not to pack more encoded data than required (see section 3.9.3).
    - Once a new input image comes in, the output signals are updated again as usual.

Given the complexity of the mathematical operations in this IP, all computations have been broken down into smaller steps to make them easier to understand and for debugging purposes as well.

Because of the two sequential clocked processes, the *Sample-Adaptive GPO2 Coder IP* only requires 2 clock cycles to generate a valid output, given a valid input.

Section 10.17 shows part of its source code.

### 3.9.2.2 Hybrid Entropy Coder IP

Figure 3.29 shows the block diagram of the *Hybrid Entropy Coder IP*:

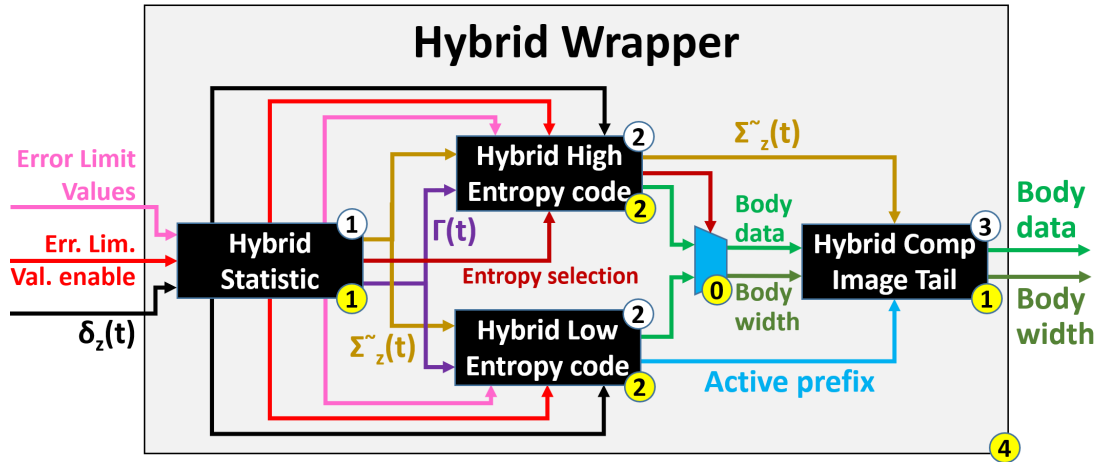


Figure 3.29: Hybrid Entropy Coder IP block diagram

This IP interconnects its sub-IPs with a minimum additional (combinational) logic in between, and it is in charge of encoding the *mapped quantizer index*  $\delta_z(t)$  and *Error Limit values* (both referred as payload) under the *Hybrid Entropy Coder* logic.

The following operations are performed, with the same execution order as listed here:

1. First, the *Hybrid Statistic IP* uses the *mapped quantizer index*  $\delta_z(t)$  values to compute the *high-resolution accumulator*  $\tilde{\Sigma}_z(t)$  and *counter*  $\Gamma(t)$  values, along with the *Entropy selection* signal (see section 3.9.2.2.1).
2. Second, both the *Hybrid High-Entropy Coder IP* and *Hybrid Low-Entropy Coder IP* take the *mapped quantizer index*  $\delta_z(t)$  and *Error Limit values* (delayed one clock cycle to be synchronized) along with the *high-resolution accumulator*  $\tilde{\Sigma}_z(t)$  and *counter*  $\Gamma(t)$  values to compute the *reverse codeword*  $R'_{k_z(t)}(\delta_z(t))$  as well as its meaningful data size (see sections 3.9.2.2.2 and 3.9.2.2.3).
3. Third, the *Entropy selection* (delayed one clock cycle to be synchronized) is used on a multiplexer to select which *reverse codeword*  $R'_{k_z(t)}(\delta_z(t))$ , from either *Hybrid High-Entropy Coder IP* or *Hybrid Low-Entropy Coder IP*, is forwarded.
4. At the end, the *Hybrid Compressed Image Tail IP* bypasses the *reverse codeword*  $R'_{k_z(t)}(\delta_z(t))$  along with its size, and only when the complete input image has been encoded, it uses the *high-resolution accumulator*  $\tilde{\Sigma}_z(t)$  and *active prefixes* (from *Hybrid Low-Entropy Coder IP*) values from all spectral bands in order to send out a final image tail data (see section 3.9.2.2.4).



As Figure 3.29 depicts, it is important to emphasize that the two available *Hybrid Entropy Coders* (*High-Entropy* and *Low-Entropy*) are working at a time.

Even though section 2.4.2.2.1 explains about using equation 2.85 to determine which *Entropy Coder* to use at every moment for encoding, here the two of them are always working, and then the multiplexer with the *Entropy selection* signal decides which incoming data is meaningful, and so, forwarded.

This approach ensures a very quick switching between *Entropy coders*, so no data is lost in the process, and the reason why *high-resolution accumulator*  $\tilde{\Sigma}_z(t)$  and *Entropy selection* signals are forwarded through the *Hybrid High-Entropy Coder IP*.

The *Hybrid Entropy Coder IP* needs a total of 4 clock cycles to produce a valid output, given a valid input.

### 3.9.2.2.1 Hybrid Statistic IP

On a very similar way to the *Sample-Adaptive Statistic IP*, the *Hybrid Statistic IP* uses the *mapped quantizer index*  $\delta_z(t)$  values on a clocked process to compute the *high-resolution accumulator*  $\tilde{\Sigma}_z(t)$  and *counter*  $\Gamma(t)$  values, according to equations 2.81, 2.82 (for initialization), 2.83 and 2.84 (for updating).

The structure of this IP is basically the same as the block diagram from Figure 3.27.

The aforementioned equations show that position  $t-1$  from *high-resolution accumulator*  $\tilde{\Sigma}_z(t)$  and *counter*  $\Gamma(t)$  values are necessary to compute the next ones, and as such previous position on time depends on the selected samples input order (see Table 3.1), 5 *Shift Register IPs* in parallel (one per input and output signal) are instantiated and configured with the right delay.

As described on section 2.4.2.2.1, every time code statistics are rescaled (condition  $\Gamma(t-1) = 2^{\gamma^*} - 1$  from equations 2.83 and 2.84), the least significant bit from  $\tilde{\Sigma}_z(t-1)$  is also outputted, so that it can be encoded in the bitstream by the next IPs.

The incoming *Error limit values* and its *enable* signal are not required in this IP either, so they are simply delayed and forwarded for the next IPs in the chain. As this delay depends on the selected samples input order, they use *Shift Register IPs* with the same configuration for such purpose.

Last but not least, after computing the new *high-resolution accumulator*  $\tilde{\Sigma}_z(t)$  and *counter*  $\Gamma(t)$  values, a non-clocked process is used to compute the next *Entropy selection* value, according to equation 2.85.

If this signal is asserted, then  $\delta_z(t)$  is a ‘*high-entropy*’ *mapped quantizer index* and the payload is encoded using the *Hybrid High-Entropy Coder IP* (see section 3.9.2.2.2). Otherwise (or simply when  $D = 2$ ),  $\delta_z(t)$  is a ‘*low-entropy*’ *mapped quantizer index* and the payload is encoded using the *Hybrid Low-Entropy Coder IP* (see section 3.9.2.2.3).

The *Shift Register IPs*, clocked process and non-clocked process from the *Hybrid Statistic IP* define a total of 1 clock cycle to generate a valid output, given a valid input.

### 3.9.2.2.2 Hybrid High-Entropy Coder IP

Almost the same as with the *Sample-Adaptive GPO2 Coder IP*, the *Hybrid High-Entropy Coder IP* takes the incoming payload (*mapped quantizer index*  $\delta_z(t)$  and *Error limit values*) to generate the *variable-length reversed GPO2 codeword*  $R'_{k_z(t)}(\delta_z(t))$ .

The structure of this IP is basically the same as the block diagram from Figure 3.28.

There are a total of two clocked processes for four different operations, executed in the same order as listed here:

1. The first clocked process uses the incoming *high-resolution accumulator*  $\tilde{\Sigma}_z(t)$  and *counter*  $\Gamma(t)$  values to compute the *variable-length code parameter*  $k_z(t)$ , operation defined on equation 2.86, and result constrained as equation 2.87 shows.
2. The second clocked process is in charge of three operations, managed by the incoming *Error Limit values enable* signal:
  - a) If the *Error Limit values enable* signal is high (just one clock cycle every  $U$  frames, see section 2.4.2), new *absolute* and/or *relative Error Limit values* are received, delayed one clock cycle to synchronize, and encoded with  $D_A$  and  $D_R$  bits, respectively.
    - The quantity of *Error limit values* to encode vary from one single value to a couple of arrays of size  $N_Z$  each, as section 3.7.2.1.1 explains.
  - b) If the *Error Limit values enable* signal is low, then the reversed GPO2 codewords  $R'_{k_z(t)}(\delta_z(t))$  are computed by using the *mapped quantizer index*  $\delta_z(t)$  values, delayed one clock cycle to synchronize, together with the *variable-length code parameter*  $k_z(t)$ , as described on section 2.4.2.2.2.
    - As this reversed codeword has a variable-length, its meaningful data-bits size must be outputted in parallel with it.
    - Only when the *Hybrid Statistic IP* performs the code selection statistics rescaling (see section 3.9.2.2.1), the LSb of  $\tilde{\Sigma}_z(t - 1)$  is also attached to the codeword, located at the beginning of it.
  - c) At the very end, once the complete image has been encoded, both output signals are fixed to 0. This small tweak makes the *Packer IP* not to pack more encoded data than required (see section 3.9.3).
    - Once a new input image comes in, the output signals are updated again as usual.

Identical to the *Sample-Adaptive GPO2 Coder IP*, all computations in this IP have been broken down into smaller steps to make them easier to understand and for debugging purposes as well.

Because of the two sequential clocked processes, the *Hybrid High-Entropy Coder IP* only requires 2 clock cycles to generate a valid output, given a valid input.

### 3.9.2.2.3 Hybrid Low-Entropy Coder IP

Figure 3.30 shows the block diagram of the *Hybrid Low-Entropy Coder IP*:

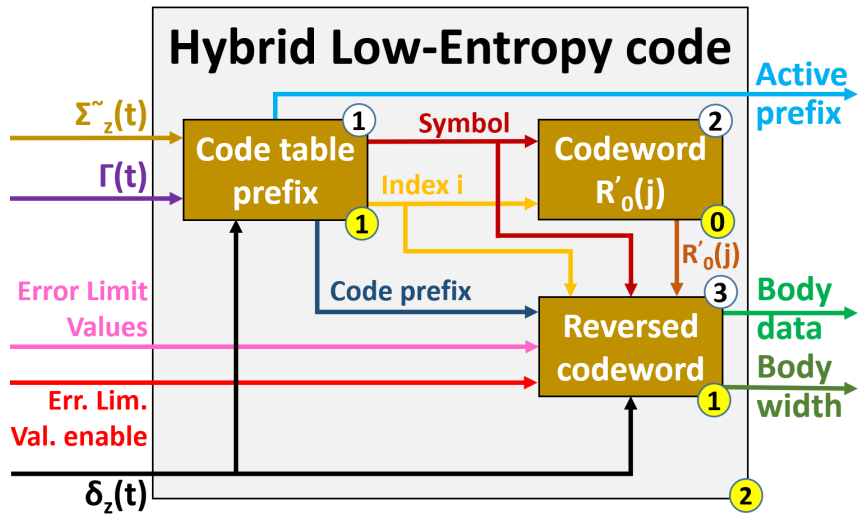


Figure 3.30: Hybrid Low-Entropy Coder IP block diagram

This IP takes the incoming payload (*mapped quantizer index*  $\delta_z(t)$  and *Error limit values*) to generate the *variable-to-variable length codeword*  $R'_{k_z(t)}(\delta_z(t))$ .

There are a total of three (mixing clocked and non-clocked) processes for eight different operations, executed in the same order as listed here:

1. The first clocked process is in charge of four operations:
  - a) The *code index table*  $i$  is computed according to equation 2.88.
  - b) The *input symbol*  $\iota_z(t)$  is computed according to equation 2.89.
  - c) With these two values, the *active prefix array* is updated and outputted.
    - The *active prefix array* is an array of 16 positions (as many as available *Hybrid Code/Flush Tables*), and inside each position there is the updated  $i^{\text{th}}$  *active prefix*.
    - Such *active prefix* update consists of attaching the new *input symbol*  $\iota_z(t)$  into the LSb position of the  $i^{\text{th}}$  column (out of 16) of the array and shifting the older ones to the right.
  - d) The new  $i^{\text{th}}$  *active prefix* is used to try to find a match (because there could be none) with the *input codeword* field from the  $i^{\text{th}}$  *Hybrid Code Table* [10, p.78-94].
    - In case there has been a match with the  $i^{\text{th}}$  *active prefix*, the  $i^{\text{th}}$  *table pointer* is stored inside the *signal code prefix*.

2. Only if the new *input symbol*  $\iota_z(t)$  turns out to be the null sequence 'X', the second non-clocked process uses the *mapped quantizer index*  $\delta_z(t)$  and *code index table*  $i$  values to compute the *residual codeword*  $R'_0(\delta_z(t) - L_i - 1)$ , in the same way as described on section 2.4.2.2.2.
3. The third clocked process is in charge of three operations, managed by the incoming *Error Limit values enable* signal:
  - a) If the *Error Limit values enable* signal is high (just one clock cycle every  $U$  frames, see section 2.4.2), new *absolute* and/or *relative Error Limit values* are received, delayed one clock cycle to synchronize, and encoded with  $D_A$  and  $D_R$  bits, respectively.
    - The quantity of *Error limit values* to encode vary from one single value to a couple of arrays of size  $N_Z$  each, as section 3.7.2.1.1 explains.
  - b) If the *Error Limit values enable* signal is low, and only when there has been a *code prefix* update in the previous clock cycle, then the *variable-to-variable length codeword*  $R'_{k_z(t)}(\delta_z(t))$  is generated by extracting the *output codeword* field from the  $i^{\text{th}}$  *Hybrid Code Table* [10, p.78-94].
    - As this reversed codeword has a variable-length, its meaningful data-bits size must be outputted in parallel with it.
    - If the new *input symbol*  $\iota_z(t)$  has been the null sequence 'X', the aforementioned *residual codeword*  $R'_0(\delta_z(t) - L_i - 1)$  shall be placed before the *output codeword* value.
    - Moreover, only when the *Hybrid Statistic IP* performs the code selection statistics rescaling (see section 3.9.2.2.1), the LSb of  $\tilde{\Sigma}_z(t - 1)$  is also attached to the codeword, located at the beginning of it.
  - c) At the very end, once the complete image has been encoded, both output signals are fixed to 0. This small tweak makes the *Packer IP* not to pack more encoded data than required (see section 3.9.3).
    - Once a new input image comes in, the output signals are updated again as usual.

Exactly the same approach as with the *Hybrid High-Entropy Coder IP*, all mathematical computations in this IP have been broken down into smaller steps to make them easier to understand and for debugging purposes as well.

The two sequential clocked processes together with the non-clocked process, make the *Hybrid Low-Entropy Coder IP* require only 2 clock cycles to generate a valid output, given a valid input.

### 3.9.2.2.4 Hybrid Compressed Image Tail IP

Figure 3.31 shows the block diagram of the *Hybrid Compressed Image Tail IP*:

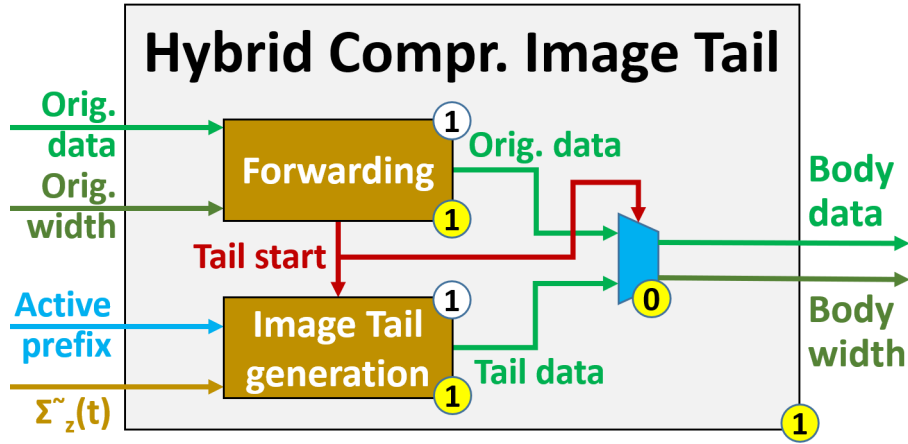


Figure 3.31: Hybrid Compressed Image Tail IP block diagram

This IP forwards the incoming encoded data (*reverse codeword*  $R'_{k_z(t)}(\delta_z(t))$ ) with its data width, and it also uses the *high-resolution accumulator*  $\tilde{\Sigma}_z(t)$  and *active prefix array* values to generate the *compressed image tail*.

There are a total of 2 clocked processes, plus combinational logic, for four different operations, executed in the same order as listed here:

1. The first clocked process is in charge of three different operations:
  - a) It forwards the incoming *reverse codeword*  $R'_{k_z(t)}(\delta_z(t))$ , along with its width.
  - b) At the end of every *spectral band*  $z$  (so when the *image coordinates*  $t = N_X * N_Y - 1$ ) the current *high-resolution accumulator*  $\tilde{\Sigma}_z(t)$  is stored into an array of  $N_Z$  positions (one per *spectral band*).
  - c) If the current *spectral band*  $z$  is the last one, so the end of the image, the incoming *active prefix array* (whose size is 16 elements as the available 16 *Hybrid Flush Tables*), is used to find a match (and there must be one) between the incoming  $i^{th}$  *active prefix* with the *active prefix* field from the  $i^{th}$  *Hybrid Flush Table*, saving the  $i^{th}$  *table pointer* when succeeded.
    - Additionally, the *tail start* flag is asserted, to inform that the incoming *reverse codeword*  $R'_{k_z(t)}(\delta_z(t))$  has been completely forwarded out.
2. The second clocked process, enabled by the previous *tail start* flag, computes and outputs the *compressed image tail* as follows, with one element per clock cycle:
  - a) In order of increasing *code index*  $i$ , the 16 previous *table pointers* are used to extract and output the 16 *flush codewords* from their corresponding *Hybrid Flush Tables*.

- Each element is outputted with its specific data width, also extracted from the very same tables.
- Next, in order of increasing *spectral band*  $z$ , all stored *high-resolution accumulator*  $\tilde{\Sigma}_z(t)$  values are also outputted, each one of them with a data width of  $2 + D + \gamma^*$  bits.
  - Finally, there is one single bit high '1' outputted, and the *tail start* flag is deasserted as well.
- At the end, there is a multiplexer (combinational logic) managed by the *tail start*, managing whether the output from the first or the second process should be outputted.

As always, all mathematical computations in this IP have been broken down into smaller steps to make them easier to understand and for debugging purposes as well.

With the two clocked processes, the *Hybrid Compressed Image Tail IP* require just 1 clock cycle to generate a valid output, given a valid input.

### 3.9.3 Packer IP

Figure 3.32 shows the block diagram of the *Packer IP*:

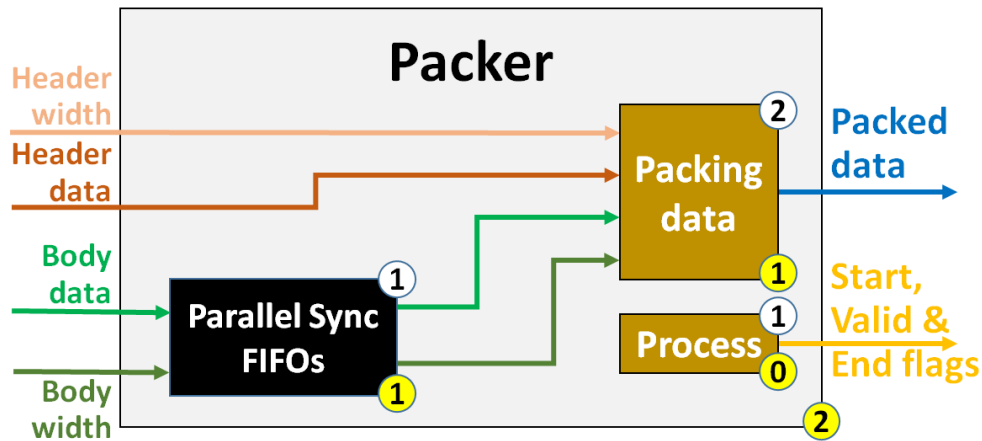


Figure 3.32: Packer IP block diagram

This IP is in charge of receiving the *Encoder Header* data and *Encoder Body* data (together with their widths) and pack them into chunks of a user-specified configurable size *output word*  $B$ . The *Encoder Header* data is always processed first, and only then, *Encoder Body* data is processed too.

Due to the fully configurable nature of the system, the *Packer IP* must be prepared to pack the data under any set-up configuration, and the first point (already mentioned on section 3.9) is the option to whether enable or disable the *Encoder Header IP*.

If such IP is disabled, the *Packer IP* has to deal only with the *Encoder Body* data, so no additional logic needs to be added. But in case the *Encoder Header IP* is indeed enabled, it turns out that the *Encoder Body* data would start coming before the *Encoder Header* data has been fully processed.

To overcome this situation, the *Parallel Synchronous FIFOs IP* is instantiated (see section 3.8.1), which stalls the incoming *Encoder Body* data and its size (plus its *enable* signal and *image coordinates*) until the *Encoder Header* data has been fully processed.

With the IP prepared not to lose any incoming data, a clocked process later is designed to pack all of it. To start packing the *Encoder Header* data, which is a very long signal with its width in parallel (static configuration, see section 3.9.1), this process reads the data chunk to chunk (size  $B$ ), and it outputs it. This is a quite straight-forward task.

But on the other side, when it is the time for the *Encoder Body* data, the data size configuration leads to one of the two following situations to occur:

- The maximum data size ( $U_{max} + D$ ) is shorter/equal than the packet size ( $B$ ).
- The maximum data size ( $U_{max} + D$ ) is longer than the packet size ( $B$ ).

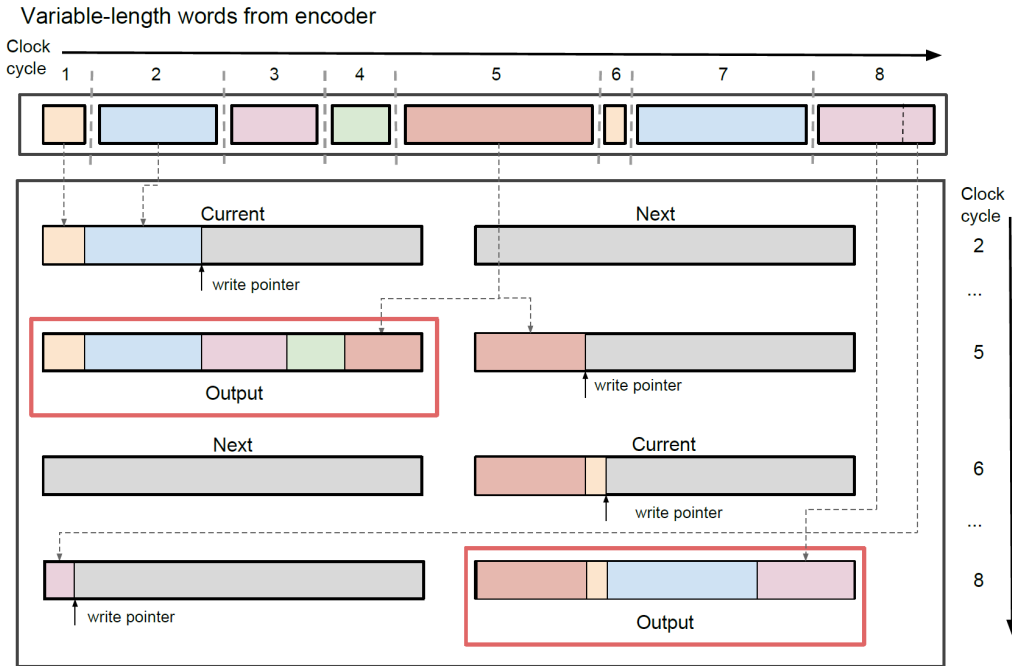


Figure 3.33: Packing data timing diagram [17, p.9]

In the first case, with *Encoder body* data (received clock after clock) shorter/equal than the packet size, only when there is enough data to fill a packet, it is outputted. If the incoming data perfectly fits the packet, then everything is outputted, but in case the last chunk does not fully fit the packet, the LSB bits that fit are sent out, and the MSB bits (overflow) are directly put into the next packet.

Figure 3.33 shows the timing diagram of packing data under this first case, and as it can be appreciated, data is written into the packet from left to right.

In the second case, with *Encoder body* data (received clock after clock) larger than what it can be packed on a single clock cycle, instead it is continuously stored on a long-enough (input image size) vector, and packets are made by accessing such vector on every clock cycle, extracting the least significant chunk, and finally shifting the remaining data to the left.

For the two cases, a bunch of counters are used to monitor every aspect of the aforementioned logic: how many valid bits are within the packets, the overflow bits to put in the next packet, data pending to be put into the packets, etc.

They are really useful, and not only for monitoring purposes, because depending on the user configuration, packing data is a task that could finish long after all input data was received, specially in the second case above (tweak already mentioned on sections 3.9.2.2.2, 3.9.2.2.2 and 3.9.2.2.3).

Finally, there is a non-clocked process to manage all flags. It uses *enable* signal and *image coordinates* to inform about the following situations:

- The first packet is sent out (*start flag*).
- A new packet is sent out (*valid flag*).
- The last packet is sent out (*end flag*).
- The packet holds header or body data (*header enable* and *body enable* flags).

This piece of logic is particularly interesting, because documentation [10, p.20] explicitly says that no logic is foreseen to inform to the outside world when a compressed image started and finished to be sent out.

Therefore, this proposal seeks to fill this void, providing a way to inform outside when the present CCSDS-123 algorithm is outputting new data, when it has started and when it has finished.

Moreover, the *end flag* is also used internally to disable part of the system until the current input image has been fully processed, and to restart everything once this moment has come (see section 3.6).

This point ensures that the algorithm will always be fully prepared to work with new data when it informs so (so no data overlapping) and the overhead or delay between input images will always be minimal (just one clock cycle).

The critical path of the *Packer IP* (when *Encoder Header IP* is enabled) defines a total of 2 clock cycles to generate a valid output, given a valid input.



# 4 Validation Plan

## 4.1 Validation scope

The present work seeks to develop a full implementation of the CCSDS-123 standard to run on FPGA, including both Issues 1 & 2, with a fully-configurable nature as its core, and for instance being able to modify the size of all signals to configure most of the implemented equations, among others.

While it is true that the whole standard (with the exception of *Block-Adaptive Entropy Coder*) has been implemented in VHDL-2008, the complete validation of such enormous system is something that goes way beyond of what a Master Thesis can cover, as the existence of such a number of configurable parameters (see Tables 10.1 to 10.4) would lead to more than 1000 configuration permutations or *test-cases*.

Therefore, only validation by simulation is performed, and focusing in the most interesting *test-cases* (as detailed in section 4.3), which makes the present implementation be the current state-of-art, such as the *Encoder Header* module or the *Periodic Error Limit Updating* option.

These *test-cases* are more than enough to prove that the current implementation is working as intended as well as to report its performance and resources utilization.

The tools for the validation plan are described on section 4.2.

It must also be mentioned that the pandemic situation with Covid-19 did not give any chance to dispose of a FPGA to work with, so this is the reason why validation by hardware (using ILAs [30] and VIOs [31] on *Xilinx Vivado IDE* over a JTAG communication) has not been possible.

## 4.2 Validation tools

Several tools have been used for the validation of the implemented source code.

Section 4.2.1 details the tool used for the validation by simulation, and section 4.2.2 the tool used for the bitstream generation and power consumption, timing and resources utilization reports.

### 4.2.1 VUnit testbenches

With a similar approach as explained in section 3.4.1 with the VHDL packages, an architecture of VHDL testbench files have been created in order to perform the validation by simulation, all of them created under the VUnit framework (see section 2.6).

The complete architecture is composed of **14 different IP levels**, grouping a total of **20 VUnit testbenches**:

1. **Top entity**: Two testbenches:
  - One for the *CCSDS-123-Issue2 Top entity IP*.
  - One for the *Parallel Synchronous FIFOs IP*.
2. **Image block**: One testbench for the *Image Coordinates Control IP*.
3. **Predictor block**: One testbench for the *Predictor Top IP*.
4. **Adder sub-block**: One testbench for the *Adder IP*.
5. **Quantizer sub-block**: One testbench for the *Quantizer IP*, *Fidelity Control IP* and *Error Limit Values Table*.
6. **Mapper sub-block**: One testbench for the *Mapper IP* and *Scaled Difference IP*.
7. **Sample Repr. sub-block**: One testbench for *Sample Representative IP*, *Clipped Quantizer Bin Center IP* and *Double-Resolution Sample Representative IP*.
8. **Prediction sub-block**: Five testbenches:
  - One for the *Prediction IP*.
  - One for the *Samples Store IP*, *Shift Register IP* and *Local Sum IP*.
  - One for the *Local Differences IP*, *Local Differences Vector IP* and *Predicted Central Local Difference IP*.
  - One for the *Weight Update Scaling Exponent IP*, *Double-Resolution Prediction Error IP* and *Weights Vector IP*.
  - One for the *High-Resolution Predicted Sample IP*, *Double-Resolution Predicted Sample IP* and *Predicted Sample IP*.
9. **Encoder block**: One testbench for the *Encoder Top IP*.
10. **Enc. Header sub-block**: One testbench for *Encoder Header IP*, *Image Metadata IP*, *Supplementary Info. Tables*, *Predictor Metadata IP* and *Encoder Metadata IP*.
11. **Enc. Body sub-block**: One testbench for the *Encoder Body IP*.
12. **Sample-Adapt. Coder sub-block**: One testbench for *Sample-Adaptive Entropy Coder IP*, *Sample-Adaptive Statistic IP* and *Sample-Adaptive GPO2 Coder IP*.
13. **Hybrid Coder sub-block**: One testbench for the *Hybrid Entropy Coder IP*, *Hybrid Statistic IP*, *Hybrid High-Entropy Coder IP*, *Hybrid Low-Entropy Coder IP* and *Hybrid Compressed Image Tail IP*.
14. **Packer sub-block**: Two testbenches:
  - One for the *Packer IP*.
  - One for the *Parallel Synchronous FIFOs IP*.

For every IP level, there is a folder called *simulation* with one or more VUnit testbenches, and even though there are not as many testbenches as IPs developed, every single one of them is instantiated in at least one testbench.

Since every VUnit testbench is composed of a Python script and a VHDL testbench, one can go to the right folder and simply execute the Python file, so that specific IP or IPs group can be tested. In such case, the executable command would be for example:

---

```
python run_adder.py
```

---

But because of the quantity of VUnit testbenches and their different locations, a bash file called *run\_simulations.sh* is created to provide a single entry point for all IP levels.

Following are the command to execute this bash file, and the GUI that prints out:

---

```
sh run_simulations.sh
```

---

```
Select a block to validate:
- Top entity: 0
- Image block: 1
- Predictor block: 2
  - Adder sub-block: 3
  - Quantizer sub-block: 4
  - Mapper sub-block: 5
  - Sample Repr. sub-block: 6
  - Prediction sub-block: 7
- Encoder block: 8
  - Enc. Header sub-block: 9
  - Enc. Body sub-block: 10
    - Sample-Adapt. Coder: 11
    - Hybrid Coder: 12
  - Packer sub-block: 13
- All blocks: 14
Clean compiled files: 15
Enter a valid number:
```

Figure 4.1: GUI of testbenches bash file

Regardless of the selected way to execute the VUnit testbench, a report with a success or fail criteria is printed out, showing that there are neither compilation nor run-time errors if successful, as Figure 4.2 shows:

```
==== Summary =====
pass vunit_lib.tb_adder.SMPL_TYPE_PY:0,SMPL_ORDER_PY:0.Top Adder Block (1.5 seconds)
pass vunit_lib.tb_adder.SMPL_TYPE_PY:0,SMPL_ORDER_PY:1.Top Adder Block (0.5 seconds)
pass vunit_lib.tb_adder.SMPL_TYPE_PY:0,SMPL_ORDER_PY:2.Top Adder Block (0.6 seconds)
pass vunit_lib.tb_adder.SMPL_TYPE_PY:1,SMPL_ORDER_PY:0.Top Adder Block (0.6 seconds)
pass vunit_lib.tb_adder.SMPL_TYPE_PY:1,SMPL_ORDER_PY:1.Top Adder Block (0.5 seconds)
pass vunit_lib.tb_adder.SMPL_TYPE_PY:1,SMPL_ORDER_PY:2.Top Adder Block (0.5 seconds)
====
pass 6 of 6
====
Total time was 4.3 seconds
Elapsed time was 4.3 seconds
====
All passed!
```

Figure 4.2: VUnit testbench report

The VUnit framework permits to introduce automatic validation criteria (by means of assertions-like functions [1]), so validation can be fully automated. Unfortunately, because of the design complexity, and specially the number of configurable parameters, this has not been doable, and instead, manual validation is performed by using the simulator engine to check the waveforms out (not in background anymore).

In this case, the *ModelSim Simulator Engine* is used [22], and Figure 4.3 is its GUI:

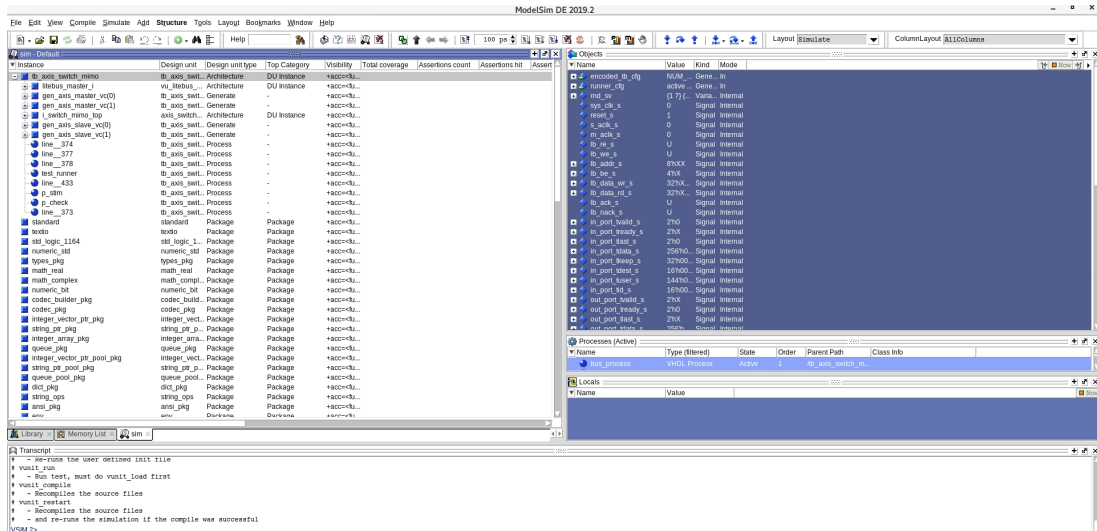


Figure 4.3: ModelSim Simulator Engine GUI

To open the *ModelSim Simulator Engine* GUI for a specific VUnit testbench, the attribute '-g' must be added when calling the Python file:

---

```
python run_adder.py -g
```

---

Sections 10.18, 10.19 and 10.20 show part of an example Python script, VUnit testbench and simulations bash file source code, respectively.

#### 4.2.2 Vivado TCL framework

With all source code successfully developed, the next step would be to open the *Xilinx Vivado IDE* tool and to generate the bitstream file to flash the FPGA.

For such purpose, a TCL framework (folder *\_vivado\_framework*) has been created, so that all steps involved in this task can be fully automated.

This TCL framework is in charge of the following tasks, executed in the same order as explained here:

1. Opens the *Xilinx Vivado IDE* tool (openable in batch and GUI modes).
2. Creates a project for the selected FPGA platform, and configures it with VHDL as the HDL to use, and default library name.

3. Adds the selected source files, and configures them with the VHDL-2008 standard.
4. Adds the selected placement and timing constraint files.
5. Executes the *Synthesis* process, with *OOC* option also configurable [35].
6. Executes the *Implementation* process [34].
7. Generates the power-consumption, utilization and timing reports.
8. Generates the bitstream to flash the FPGA.

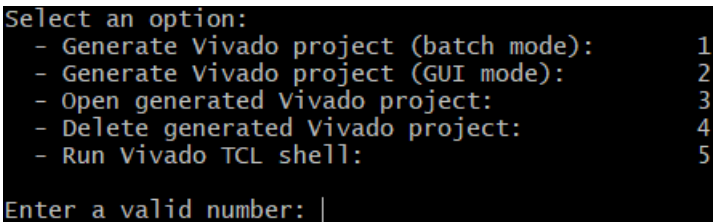
To make this framework more accessible and easier to manage, a bash file called *run\_vivado\_project.sh* is created to make use of it.

It is executed with the following command, and Figure 4.4 is the GUI that prints out:

---

```
sh run_vivado_project.sh
```

---



```
Select an option:
- Generate Vivado project (batch mode): 1
- Generate Vivado project (GUI mode): 2
- Open generated Vivado project: 3
- Delete generated Vivado project: 4
- Run Vivado TCL shell: 5
Enter a valid number: |
```

Figure 4.4: GUI of Xilinx Vivado bash file

As the previous figure clearly shows, the Xilinx Vivado bash file accesses the TCL framework to provide the following options:

1. Generate a Vivado project under batch mode.
2. Generate a Vivado project under GUI mode.
3. Open the already generated Vivado project.
4. Close the already generated Vivado project.
5. Execute the Vivado TCL shell.

The generated project will always be called *ccsds123issue2\_project* and located inside the folder *\_vivado\_framework*, and all generated output files (bitstream and reports) are copied into folder *artifacts*, also located inside the same folder.

Last but not least, and even though it is not expected, one can modify the TCL framework by going to the folder *\_vivado\_framework* and accessing the TCL files. This allows to add/remove source, constraint files and block designs, do not execute *Synthesis* or *Implementation*, change the FPGA target device, etc.

Sections 10.21 and 10.22 show part of the TCL framework and Xilinx Vivado bash file source code, respectively.

## 4.3 Test-cases

The present algorithm has been designed with a fully configurable nature as its core, being every single IP instantiation dependent to the user configuration, and some of them even not instantiated if configured so, such as the *Encoder Header IP*. These dependencies are explained in chapter 3.

All these configuration possibilities make impractical validate every single *test-case*, as there would be way more than 1000 of them (or configuration permutations). Check section 3.5.1 and Tables 10.1, 10.2, 10.3, 10.4 to understand the magnitude of it.

Therefore, a bunch of *test-cases* must be selected to perform the validation, and still provide results good enough to demonstrate that the design is working as intended.

Using the same approach as how the source code has been structured (see section 3.5), the *test-cases* are sorted out by the following groups:

- *Image Coordinates Control* block: Results are displayed on section 5.3.1.
- *Predictor* block: Results are displayed on section 5.3.2.
- *Encoder* block: Results are displayed on section 5.3.3.
- *Top Entity* block: Results are displayed on section 5.3.4.

For each group, first all IPs are validated in an isolated way (putting special emphasis in the ones that give added value to this implementation), then a validation scales up to the major blocks, to finish validating the very top entity.

This statement just applies to the functionality aspect of the system, as the parameters configuration will remain (almost) untouched because of the reason given above.

Nevertheless, the following 'standardized' *test-case* is the one used for providing the results of the bitstream generation as well as HW integration reports, discussed on sections 5.1 and 5.2, respectively:

- Issue 1 configuration (see Table 2.18).
- Signed input samples with a data range of 16 bits.
- BIP input order.
- *Full prediction mode* selected.
- Lossless compression selected.
- *Sample-Adaptive Entropy Coder IP* selected.
- *Encoder Header IP* disabled.
- *Periodic Error Limit Updating* option disabled.

# 5 Results

## 5.1 Bitstream generation

Using the 'standardized' *test-case* (see section 4.3), passing the *Synthesis* and *Implementation* processes has been quite fast, less than **20 minutes** altogether. Indeed, any other configuration does not require much more (or less) time than that, not even enabling the OOC option (see section 2.7), so this value is a good average.

This is due to the fact that the presented algorithm is a pure PL (FPGA design using *hdl*), and absolutely nothing from PS (uC design using SW prog. languages). Moreover, as the title of document [10] suggests, the CCSDS-123 standard has a low-complexity.

However, it is interesting to mention that the time to generate the bitstream is substantially increased when either there are time violations (*Vivado IDE* spends more time on looking for alternative routing paths that fulfils the required timing) or the design demands almost all resources within the FPGA (a lack of resources makes really difficult for *Vivado IDE* to find a single routing path).

In fact, FPGAs invest way more resources on interconnecting the IPs, rather than implementing the IPs themselves, and this relation is usually about 90%-10% [20], which makes the previous statement quite understandable.

With the bitstream already generated, *Vivado IDE* reports several *critical warnings*, all of them because the *placement constraints* file has constrained the input image signal width as 32-bits (maximum possible case), when the current user configuration is using a width of 16-bits (user-specified parameter *dynamic range D*). Thus, the constraints file is already prepared for any user configuration, and these should simply be ignored, so 0 *critical warnings* can be assumed here.

If the schematics generated after *Synthesis* and *Implementation* processes are compared, the first one infers the design using a total of **642 cells, 87 I/O ports and 1590 nets**, and the second one infers the design with a total of **641 cells, 87 I/O ports and 1550 nets**.

Even though the numbers are quite similar for both processes, this comparison clearly demonstrates that the *Vivado IDE* can perform more resources optimizations once the target device is known, as stated on section 2.8.

Using the TCL framework, located within folder *\_vivado\_framework*, along with its associated bash file (see section 4.2.2), the generation of this bitstream is automated and stored into folder *artifacts*, also inside folder *\_vivado\_framework*.

## 5.2 HW integration reports

This section shows and discusses the reports generated with *Vivado IDE*, after passing both *Synthesis* [35] and *Implementation* [34] processes.

As already mentioned on sections 2.7 and 2.8, the existing differences for each report between these two processes are because the *Synthesis* just provides an estimation of the final design, and the *Implementation* provides an exact result (and usually better), because only the second one knows the target device, and so, it can take benefit of the available FPGA resources.

It must be mentioned once again that these results correspond to 'standardized' *test-case* from section 4.3, and as this algorithm is fully configurable, they would slightly change when a new user configuration is defined.

### 5.2.1 Power consumption report

Both power consumption reports after passing *Synthesis* and *Implementation* processes are displayed on Figure 5.1. Overall, both report almost the same power consumption: **0.713W** in the first case (left), and **0.721W** in the second case (right).

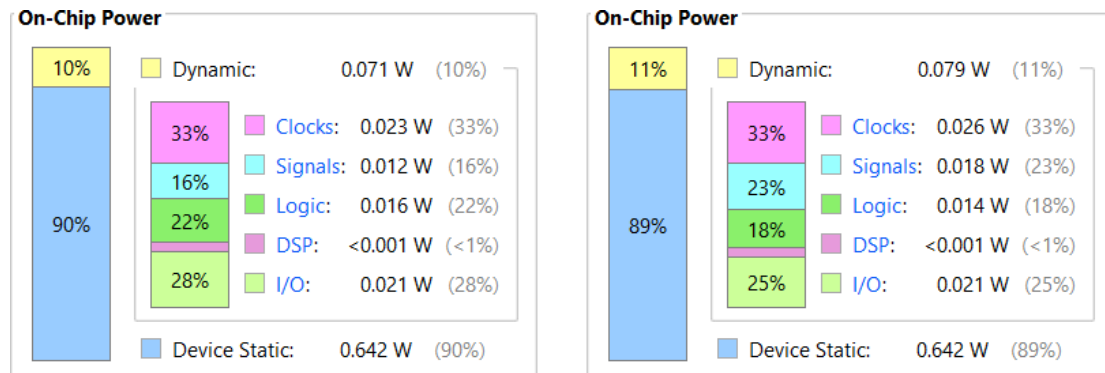


Figure 5.1: Power consumption reports for *Synthesis* (left) and *Implementation* (right)

Regardless of the final value, it is quite interesting to see that 10% corresponds to *dynamic power consumption* (components toggling continuously during normal functionality), and the other 90% corresponds to *static power consumption* (leakages from capacitors while keeping the data).

These results say that the design is continuously storing a lot of data, much more than this is being computed. This is very likely due to the quantity of *Shift Register IPs* required in the *Predictor* block to manage its *neighbour input samples* and the two existing close-loop branches on it.

Anyway, this low *dynamic power consumption* is also thanks to the set frequency of the clock signal (see section 5.2.1), as keeping the frequency as slow as possible is translated into a decrease in the number of times per second that all components toggle [15].



## 5.2.2 Utilization report

Figures 5.2 and 5.3 list the Xilinx primitive resources used to infer the presented algorithm (and also compared with the available ones) after passing *Synthesis* and *Implementation* processes, respectively:

Resource	Estimation	Available	Utilization %
LUT	11986	274080	4.37
LUTRAM	77	144000	0.05
FF	25258	548160	4.61
DSP	1	2520	0.04
IO	87	328	26.52
BUFG	1	404	0.25

Figure 5.2: Reduced Utilization report for *Synthesis*

Resource	Utilization	Available	Utilization %
LUT	11964	274080	4.37
LUTRAM	69	144000	0.05
FF	25258	548160	4.61
DSP	1	2520	0.04
IO	87	328	26.52
BUFG	2	404	0.50

Figure 5.3: Reduced Utilization report for *Implementation*

One more time, here there are (almost) no improvements between the 2 processes, yet just a minimal decrease in the number of LUTs and LUTRAMs, which do not even modify the *Utilization %* field.

In any case, the percentage of utilization is very small compared with what *Zynq Ultra-Scale+ MPSoC ZCU102 Evaluation Board* offers, and this ensures that other algorithms can work together with the CCSDS-123 algorithm under the same hardware platform.

Going a bit deeper, Figures 10.2 and 10.3 show the extended versions of the previous utilization reports, where not only more types of Xilinx primitives are listed, but they also show the utilization required for every single (instantiated) IP.

It can be seen that the *Encoder* block demands more logic resources than the *Predictor* block (which has more and simpler sub-IPs), something totally normal due to the fully configurable nature of the *Encoder* block, always changing the width of its output signal.

However, the most demanding one is the *Adder IP*, since it manages the big close-loop branch from the *Predictor* block, and it must store all *original samples*  $s_z(t)$  and *predicted samples*  $\hat{s}_z(t)$  to compute the *prediction residual*  $\Delta_z(t)$ , demanding a lot of FFs for it.

Finally, in case the available FPGA resources were not enough to implement the design, the *Vivado IDE* tool provides advanced options to highlight which RTL modules or routing paths consume more resources (and what kind of) specifically, so that an optimization of the conflicted parts of the design could be executed [32][34].

### 5.2.3 Timing report

Figures 5.4 and 5.5 show the timing report of the present design, after passing *Synthesis* and *Implementation* processes, respectively:

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 2,708 ns	Worst Hold Slack (WHS): -0,058 ns	Worst Pulse Width Slack (WPWS): 11,968 ns
Total Negative Slack (TNS): 0,000 ns	Total Hold Slack (THS): -36,205 ns	Total Pulse Width Negative Slack (TPWS): 0,000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 3381	Number of Failing Endpoints: 0
Total Number of Endpoints: 42898	Total Number of Endpoints: 42898	Total Number of Endpoints: 25334

**Timing constraints are not met.**

Figure 5.4: Timing report for Synthesis

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 0,311 ns	Worst Hold Slack (WHS): 0,003 ns	Worst Pulse Width Slack (WPWS): 11,968 ns
Total Negative Slack (TNS): 0,000 ns	Total Hold Slack (THS): 0,000 ns	Total Pulse Width Negative Slack (TPWS): 0,000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 42898	Total Number of Endpoints: 42898	Total Number of Endpoints: 25334

**All user specified timing constraints are met.**

Figure 5.5: Timing report for Implementation

As it can be appreciated, the timing constraints were not met after executing *Synthesis*, failing in the *Hold time* [15] part. Nonetheless, such violations are automatically fixed after executing *Implementation*, and this is thanks to the extra optimizations done in the routing paths, only possible because the FPGA target device is known here [34].

Unfortunately, the clock frequency had to be dropped to **40MHz** to meet all the timing constraints, instead of the original idea of **100-125MHz**.

This unexpected event has occurred because the implemented equations have not been broken down properly. Chapter 3 says many times that the equations have been broken down into several steps, but that was using *variables* (so most things done at a time), so that the readability of the source code improves, but it should have been done using *signals* instead (so one step done per clock cycle). Such modification makes most of the IPs require more clock cycles to produce a valid output, given a valid input, but on the other hand, they can work at a higher frequency.

When there were timing violations, the critical path of the design was in the *Encoder* block, in this case the *Sample-Adaptive Entropy Coder*, due to the fact that it is the most demanding part of the implementation.

Moreover, the timing violations were sorted out as *Intra-Clock Paths* because the whole system is using just one clock signal, so no different clock domains to deal with [15].

Despite this unexpected result, the timing performance is still quite good, and fixing the code to break down the equations properly would be really easy to perform, since the source code architecture has already been designed thinking in this kind of eventualities (see section 3.4).

## 5.3 Functionality outcome

While the design of the source code has followed a top-down approach (see chapter 3), its validation has followed a bottom-up approach. In other words, the design started from a high-level point of view, making first the IPs architecture, to a low-level point of view, where they are implemented; and validation started checking all IPs in an isolated way, to finally perform the integration tests.

This is the *V-Model methodology*, created to reduce the working effort, and to maximize the flexibility and results [12].

Therefore, sections 5.3.1, 5.3.2, 5.3.3 and 5.3.4 detail the isolated and integration tests that have been carried out, and their respective results.

Such tests have been performed using the testbenches architecture, created with the VUnit framework, as detailed on section 4.2.1.

Both isolated and integration tests consist on visual validation by checking the waveforms, and additionally for the very top entity (see section 5.3.4), results are also compared with the VHDL implementation from reference [17].

Anyway, because of the huge quantity of *signals* and configuration possibilities in the design, it is almost impossible to attach a waveform of every one of them. Instead, results are simply explained, and only waveforms showing the most interesting parts are attached here.

As explained on section 3.4, and applicable to the whole source code, all waveforms below demonstrate that no *signal* nor *variable* have neither open-circuit nor short-circuit state in any part of the design.

### 5.3.1 Image Coordinates Control IP block

The *Image Coordinates Control IP*, the only one within this block, works as intended, counting the coordinates depending on the selected samples input order, and all output *signals* are deasserted once the input *signal image\_end* is asserted.

Figure 5.6 is the waveform showing the expected behaviour, in this particular case with the BIP order:

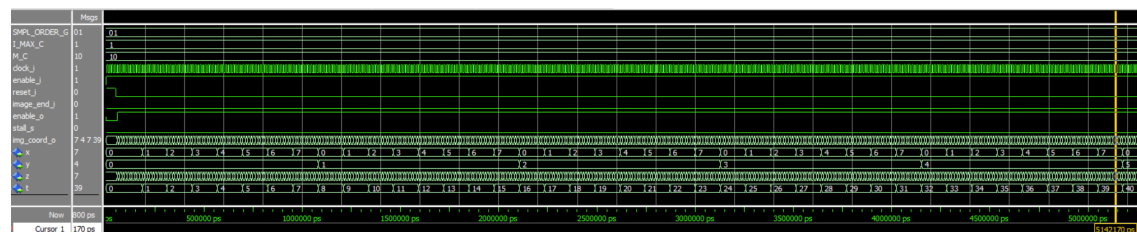


Figure 5.6: Image Coordinates Control IP waveform

### 5.3.2 Predictor IP block

#### 5.3.2.1 Adder IP sub-block

The *Adder IP*, the only one within this sub-block, works as expected, and Figure 5.7 shows its waveform:

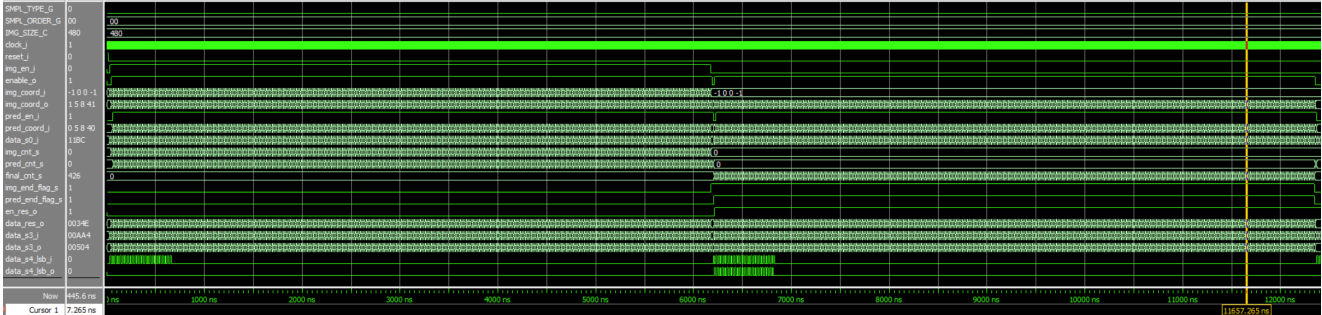


Figure 5.7: Adder IP waveform

As explained on section, this IP successfully outputs all *original samples*  $s_z(t)$  to produce the *predicted samples*  $\hat{s}_z(t)$ , and once both are fully received and stored (middle point from Figure 5.7), it outputs now the synchronized *prediction residual*  $\Delta_z(t)$ .

Such behaviour, which involves storing a lot of data and outputting the whole image twice, explains why Figures 10.2 and 10.3 say that this IP requires a lot of resources (mainly FFs).

#### 5.3.2.2 Quantizer IP sub-block

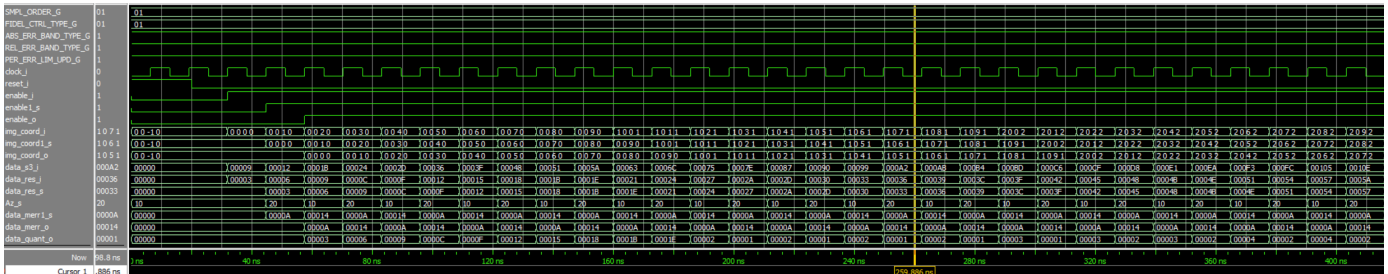


Figure 5.8: Quantizer IP waveform

Only involving the *Quantizer IP*, *Fidelity Control IP* and *Error Limit Values Table*, Figure 5.8 shows the waveform of the *Quantizer* sub-block.

Values are computed as expected, and the different *enable* signals and *image coordinates* clearly define when the sub-IPs start working.

In this case, *Periodic Error Limit Updating* with *absolute error limit* option is enabled, so one can see how *Error limit values* (and *maximum error*  $m_z(t)$  too) changes over time.

### 5.3.2.3 Mapper IP sub-block

Figure 5.9 is the waveform of the *Mapper* sub-block, covering the *Mapper IP* and the *Scaled Difference IP*:

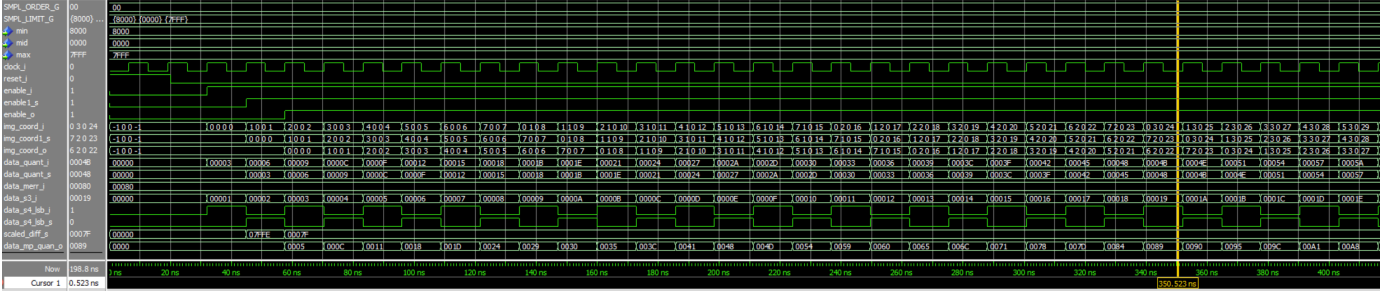


Figure 5.9: Mapper IP waveform

This sub-block compute the *signed quantizer index*  $q_z(t)$  and *scaled difference*  $\theta_z(t)$  as intended, and properly synchronized by means of the *enable* signal and *image coordinates*.

### 5.3.2.4 Sample Representative IP sub-block

The waveform from the *Sample Representative* sub-block is shown in Figure 5.10, including the *Sample Representative IP*, *Clipped Quantizer Bin Center IP* and *Double-Resolution Sample Representative IP*:

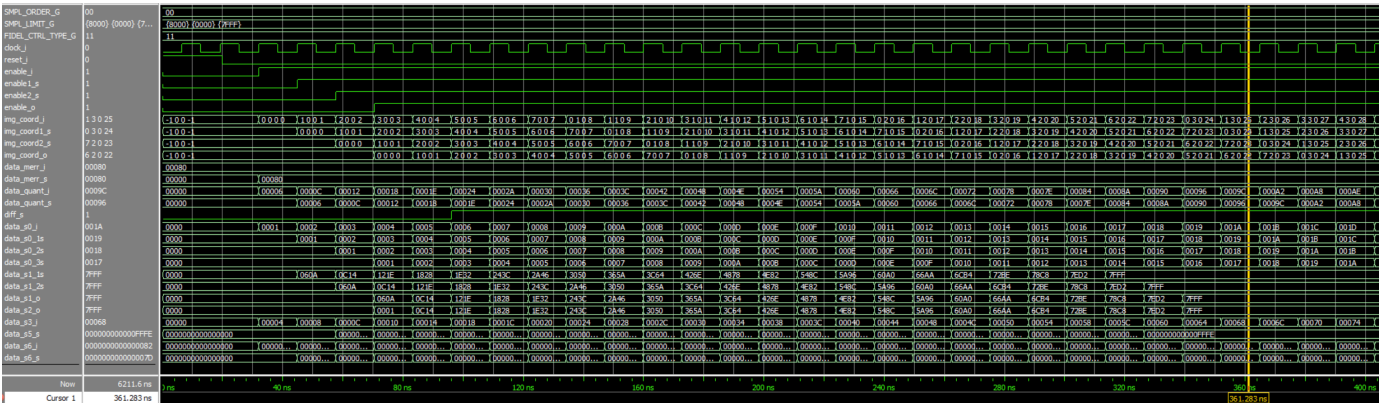


Figure 5.10: Sample Representative IP waveform

The figure above shows clearly all synchronization signals among the sub-IPs, which consist on delay signals (such as *original sample*  $s_z(t)$  or *clipped quantizer bin center*  $s'_z(t)$ ) in order to provide the right value together with the right *image coordinates*.

All values are computed as expected and forwarded to the output properly synchronized, with no data lost in between.

### 5.3.2.5 Prediction IP sub-block

The *Prediction* sub-block includes a total of 12 sub-IPs: *Samples Store IP*, *Shift Register IP*, *Local Sum IP*, *Local Differences IP*, *Local Differences Vector IP*, *Weight Update Scaling Exponent IP*, *Double-Resolution Prediction Error IP*, *Weights Vector IP*, *Predicted Central Local Difference IP*, *High-Resolution Predicted Sample IP*, *Double-Resolution Predicted Sample IP* and *Predicted Sample IP*.

Figure 5.11 shows the waveform including all these IPs:

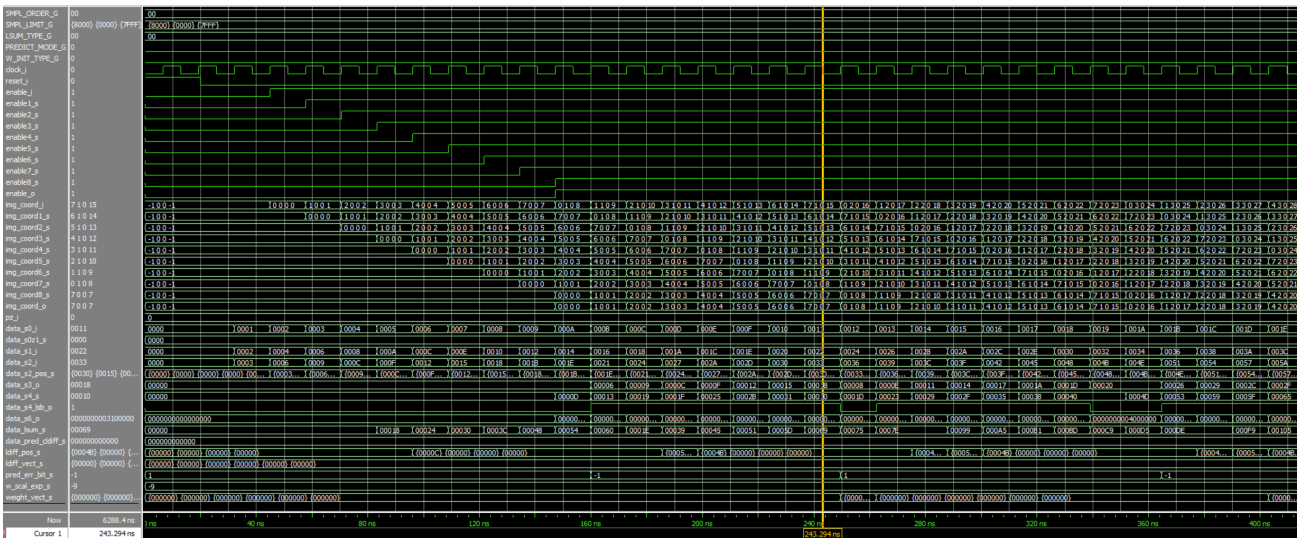


Figure 5.11: Prediction IP waveform

This test shows that all values are computed as intended, and more importantly, no data is lost along this long path, which even includes a close-loop branch.

Here, a lot of *enable* signals and *image coordinates* can be seen, which successfully control the synchronization of such 12 sub-IPs.

### 5.3.2.6 Predictor Top IP integration test

The *Predictor Top IP*, where sections 5.3.2.1, 5.3.2.2, 5.3.2.3, 5.3.2.4 and 5.3.2.5 are included, simply consisted on seeing that all data is flowing as expected, given that if the sub-IP can compute their values in a isolated test, here it should be the same.

The synchronization among all modules is correct and values are computed as expected, even with the *Periodic Error Limit Updating* option enabled, where *maximum error*  $m_z(t)$  changes over time.

And in the same way as discussed with the *Adder IP*, this test shows the whole input image is forwarded twice: one to generate the *predicted sample*  $\hat{s}_z(t)$ , and using such signal, the proper *prediction residual*  $\Delta_z(t)$  can be generated, computing the final *mapped quantizer index*  $\delta_z(t)$ .

Figure 5.12 is the waveform of the *Predictor Top IP*.

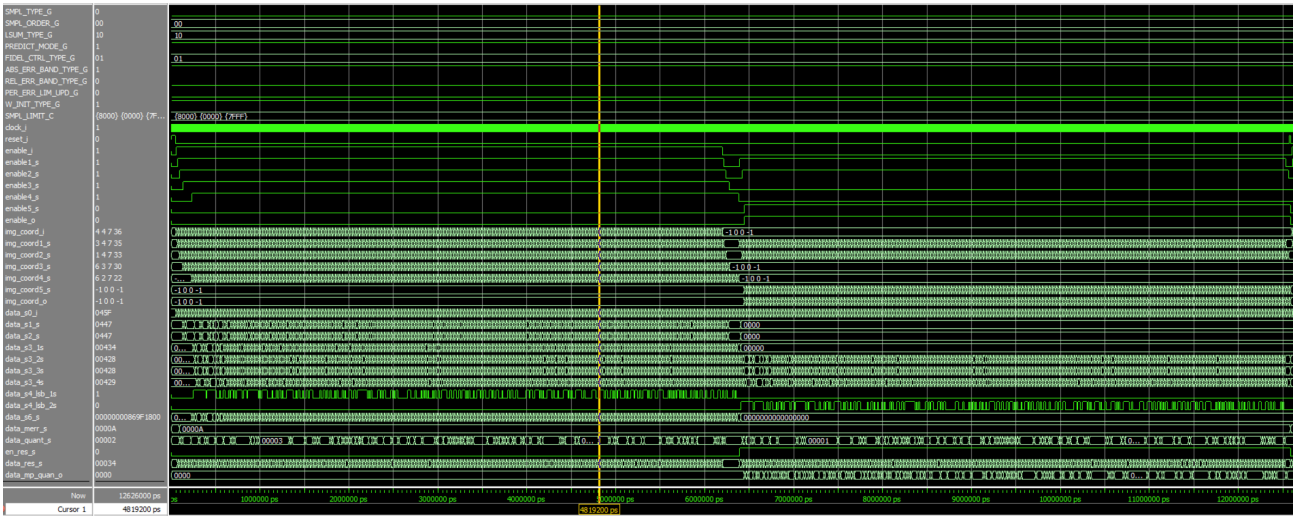


Figure 5.12: Predictor Top IP waveform

### 5.3.3 Encoder IP block

#### 5.3.3.1 Encoder Header IP sub-block

Figure 5.13 is the waveform of the *Encoder Header IP*. As this IP is fully combinational logic (so payload is generated immediately), there is no need to see a graph with signals changing over time (because they do not), but instead, just see the initial values:



Figure 5.13: Encoder Header IP waveform

This figure shows that *Image*, *Predictor* and *Encoder Metadata* are generated according to the configuration above, and they are saved on very long signals, with their respective useful data-width on separated signals.

Then, one can see that such data is joined together in the final *Encoder Header metadata*, concatenating only the meaningful bits, and again, the new data is stored on a very long signal, with another signal in parallel detailing the useful data-width.

### 5.3.3.2 Sample-Adaptive Entropy Coder IP sub-block

Figure 5.14 shows the waveform of the *Sample-Adaptive Entropy Coder sub-block*, covering the *Sample-Adaptive Entropy Coder IP*, *Sample-Adaptive Statistic IP* and *Sample-Adaptive GPO2 Coder IP*:

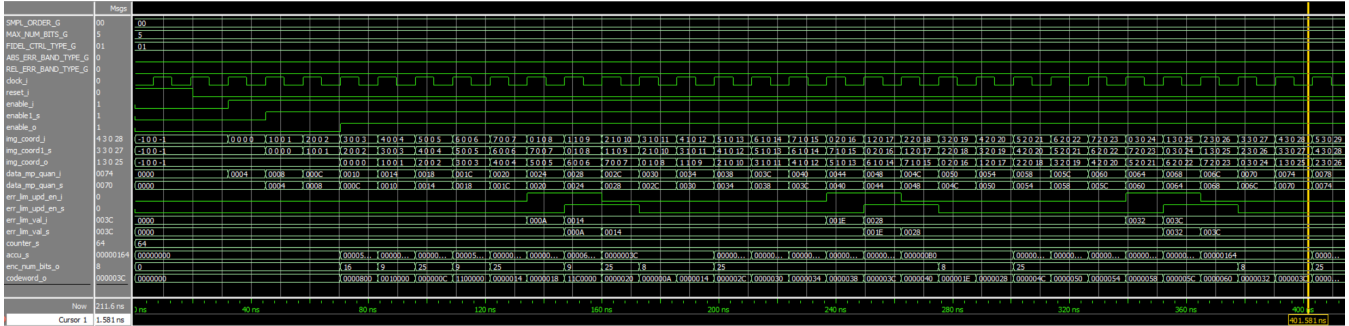


Figure 5.14: Sample-Adaptive Entropy Coder IP waveform

The values are properly computed, encoding the payload as expected, and synchronized among the sub-IPs, so no data is lost.

Moreover, it can be seen that apart from the *mapped quantizer index*  $\delta_z(t)$ , new *Error Limit values* (with its associated *enable* signal) are periodically introduced after some clock cycles, at which point they are encoded as well.

As the output codeword is variable-length too, the waveform shows such payload is stored into a fixed-size signal, and another signal in parallel with its useful data-width.

### 5.3.3.3 Hybrid Entropy Coder IP sub-block

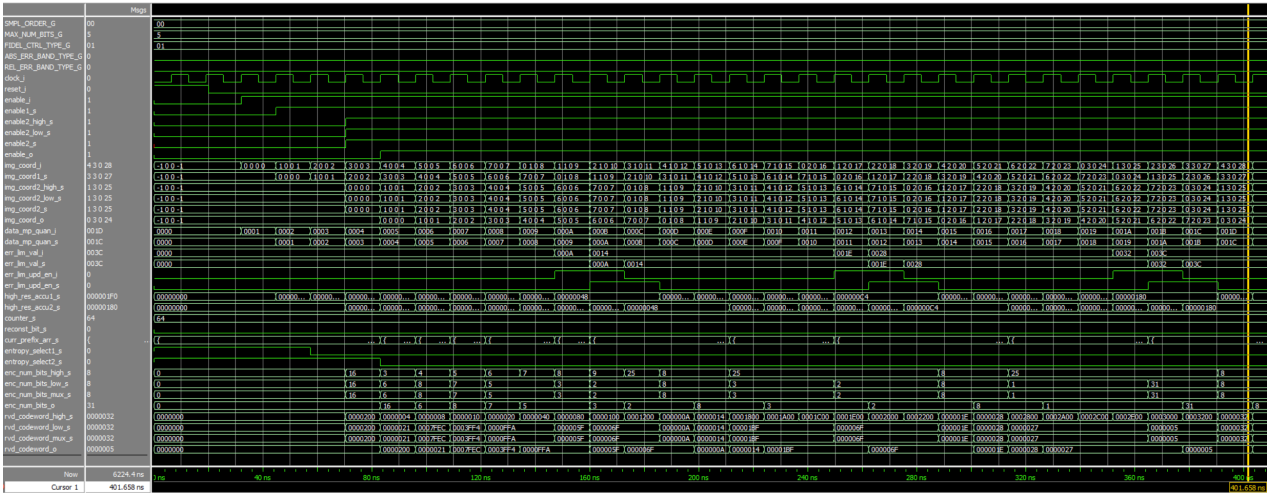


Figure 5.15: Hybrid Entropy Coder IP waveform



Figure 5.15 is the waveform of the *Hybrid Entropy Coder sub-block*, covering the *Hybrid Entropy Coder IP*, *Hybrid Statistic IP*, *Hybrid High-Entropy Coder IP*, *Hybrid Low-Entropy Coder IP* and *Hybrid Compressed Image Tail IP*.

On a very similar way as with the *Sample-Adaptive Entropy Coder sub-block*, here the *mapped quantizer index  $\delta_z(t)$*  are encoded as expected, and when new *Error Limit values* are received, they are encoded too. Moreover, the output variable-length codeword is also generated along with its useful data-width in parallel.

It turns out the *Hybrid Entropy Coder sub-block* is not synthesizable, more concretely due to the *Hybrid Low-Entropy Coder IP*, which is using several array of strings to handle the *Hybrid Code/Flush Tables*. To overcome this problem, a small update by defining a 'string to std\_logic\_vector' conversion would be necessary, redefining such tables.

### 5.3.3.4 Packer IP sub-block

Figure 5.16 is the waveform for *Packer FIFOs IP* and *Parallel Synchronous FIFOs IP*:

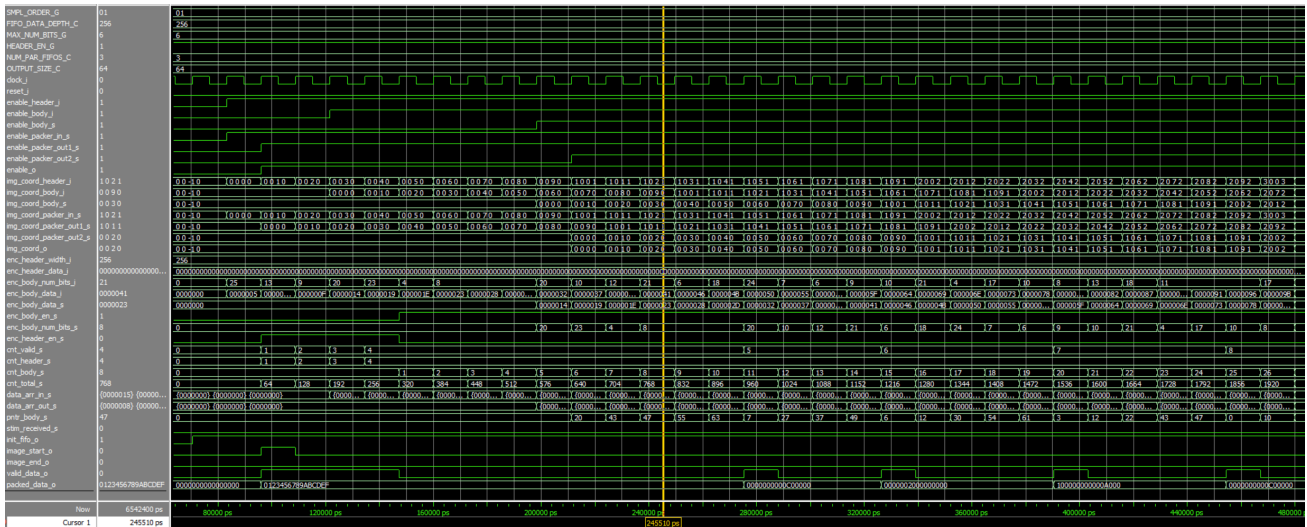


Figure 5.16: Packer IP waveform

This configuration both *Encoder Header* and *Encoder Body* data comes in together, each one with their respective data-width in parallel, and they are packed and outputted as intended. It always starts packing with the *Encoder Header*, and then it moves to the *Encoder Body*, filling the packet with 0-padding in the last iteration for both cases.

Although the *Encoder Body* data was already coming in while packing the *Encoder Header* data, it can be seen that no data was lost, thanks to the *Parallel Synchronous FIFOs*, holding the data until ready to be processed.

Furthermore, the flags to know when the packing started, finished, new packet created, FIFOs are ready, header or body data are being outputted, also work as expected.

### 5.3.3.5 Encoder Top IP integration test

Figure 5.17 shows the waveform of the *Encoder Top IP*:

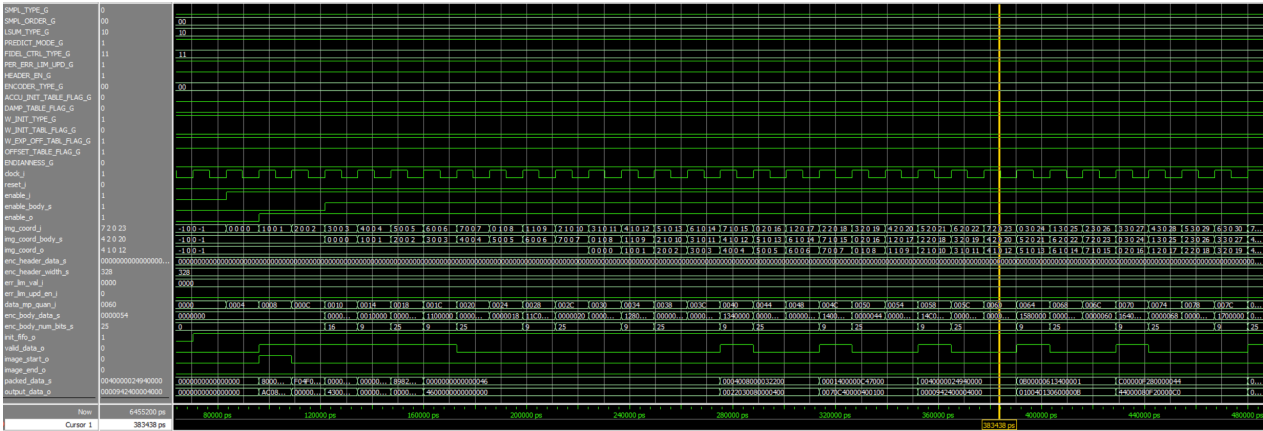


Figure 5.17: Encoder Top IP waveform

The *Encoder Top IP*, where sections 5.3.3.1, 5.3.3.2, 5.3.3.3 and 5.3.3.4 are included, simply consisted on seeing that all data is flowing as expected, given that if the sub-IP can compute their values in a isolated test, here it should be the same.

The synchronization among all modules is correct, and all values are computed as expected, generating packets out from the *Encoder Header* and *Encoder Body* data, with the corresponding flags asserted properly.

Additionally from the *Encoder Top IP*, the generated packets are also properly re-ordered, byte to byte, depending on the *Endianness* configuration.

### 5.3.4 Top Entity IP integration test

Figure 5.18 shows the waveform of the *CCSDS-123 Top Entity IP*. This block includes not only all IPs tested above, but also the *Predictor-Encoder Interconnection IP*, which connects the *Predictor* and *Encoder* blocks.

Once the *Predictor IP* outputs the *mapped quantizer index*  $\delta_z(t)$ , such data is forwarded into the *Encoder IP*, swapping with the *Error Limit Values* every time they are updated. Finally, the *Encoder IP* outputs the data packets together with all flags as intended.

Working with this very *CCSDS-123 Top Entity IP*, not only the waveform from Figure 5.18 was used to do a visual validation of the mathematical operations and synchronization signals, but something else was carried out. Using the 'standardized' *test-case* configuration from section 4.3, the source code from reference [17] was also used to compare the output signals from the two implementations.

**Both projects output the same quantity of packets and with the same payload**, under several different user configurations.

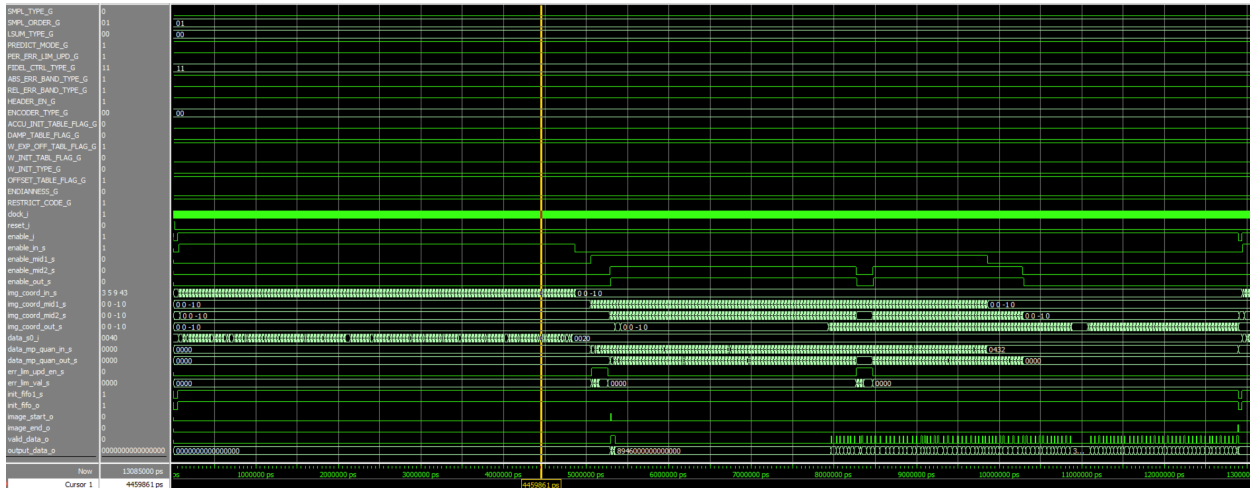


Figure 5.18: Top Entity IP waveform

## 5.4 Performance & Final results

As just said on section 5.3, the presented CCSDS-123 Issues 1 & 2 algorithm is fully working as intended, at least in all configured *test-cases*, because there are too many configuration permutations to test every single one of them.

Always using the waveforms, the tests were carried out by both visual validation of *signals* (applicable for all IPs on an isolated way, plus integration tests), and by comparing with the VHDL implementation from reference [17] too (applicable for *Top Entity IP*).

Apart from functionality, these tests were also used to compute the **total clock cycles** ( $Total_{CC}$ ) to fully compress an incoming image, reflected on equations 5.1, 5.2 and 5.3:

$$Predictor_{CC} = (N_X * N_Y * N_Z) + (2 * (1 + 2 + 2)) + (8 + 3) \quad (5.1)$$

$$Encoder_{CC} = 0 + (4 + 6) + 2 \quad (5.2)$$

$$Total_{CC} = (N_X * N_Y * N_Z) + Predictor_{CC} + Encoder_{CC} \quad (5.3)$$

$N_X$ ,  $N_Y$  and  $N_Z$  are the configured *image dimensions*, and the numbers refer to the time that both blocks need to compute their respective output signals (see chapter 3).

In the particular case of the *Predictor* block, its equation shows how the input image is fully forwarded once ( $N_X * N_Y * N_Z$ ), traveling the whole block ( $1 + 2 + 2 + 8 + 3$ ) to finally cross the open-loop one more time ( $1 + 2 + 2$ ), in order to generate the final *prediction residual*  $\Delta_z(t)$  (see section 3.7.1) and then *mapped quantizer index*  $\delta_z(t)$ .

Nevertheless, the  $Total_{CC}$  value could slightly change depending on the *image compression ratio* that the specific user configuration provides.

After several tests, the measured *image compression ratio* is between the **40% and 60% of the original input image size**, and such fluctuations depend on the provided user configuration, mainly by the user-specified parameter *output word size*  $B$ .

## 5.5 Time planning

The system introduced in this report is the work of 15 months. It started as a *Specialization Project* [24], scheduled from 28/08/2020 to 19/12/2020, and then it continued with the present *Master Thesis*, scheduled from 10/01/2021 to 18/01/2022.

The time planning (*Specialization Project* + *Master Thesis*) was structured into the following points:

1. **Packages:** It comprises the development of the VHDL packages described in section 3.4.1. It took an approximated time of **160 hours**.
2. **IPs development:** It comprises the creation of every single IP, covering sections 3.5 to 3.9.3, as well as their respective VUnit testbenches to perform their validation. It took an approximated time of **1000 hours**.
3. **Simulations Bash file:** It comprises the bash file to unify and execute all VUnit testbenches. It took an approximated time of **80 hours**.
4. **Xilinx Vivado Bash file:** It comprises the bash file, together with the TCL framework, to automate the creation of the Xilinx Vivado project, along with the bitstream and reports generation. It took an approximated time of **100 hours**.
5. **IPs integration:** It comprises the constraint files declaration and all necessary corrections in the source code files, so that there are no timing violations and the bitstream can be successfully created. It took an approximated time of **60 hours**.
6. **Documentation:** It comprises writing the present report and all documentation research associated with it, such as exploring the current published papers about the CCSDS-123 standard. It took an approximated time of **400 hours**.

Therefore, the total time invested into this project has been an approximated amount of **1800 hours**.

Figure 5.19 represents the Gantt chart for this time planning, and some interesting points can be extracted from it.

- Although the *Predictor* block integrates many sub-IPs, the approach of one IP per mathematical operation supposed a lot of small IPs that were developed quite fast.
  - This applies only to the easy blocks, as some others (like the *Weights vector IP*) were a bit difficult to fully understand them prior to be implemented.
- The *Encoder* block required more effort to be developed than the *Predictor* block, even though the second one is simpler than the first one.
- Obviously, the write of this report took quite a lot of time, so it slowed down the source code development and its validation.

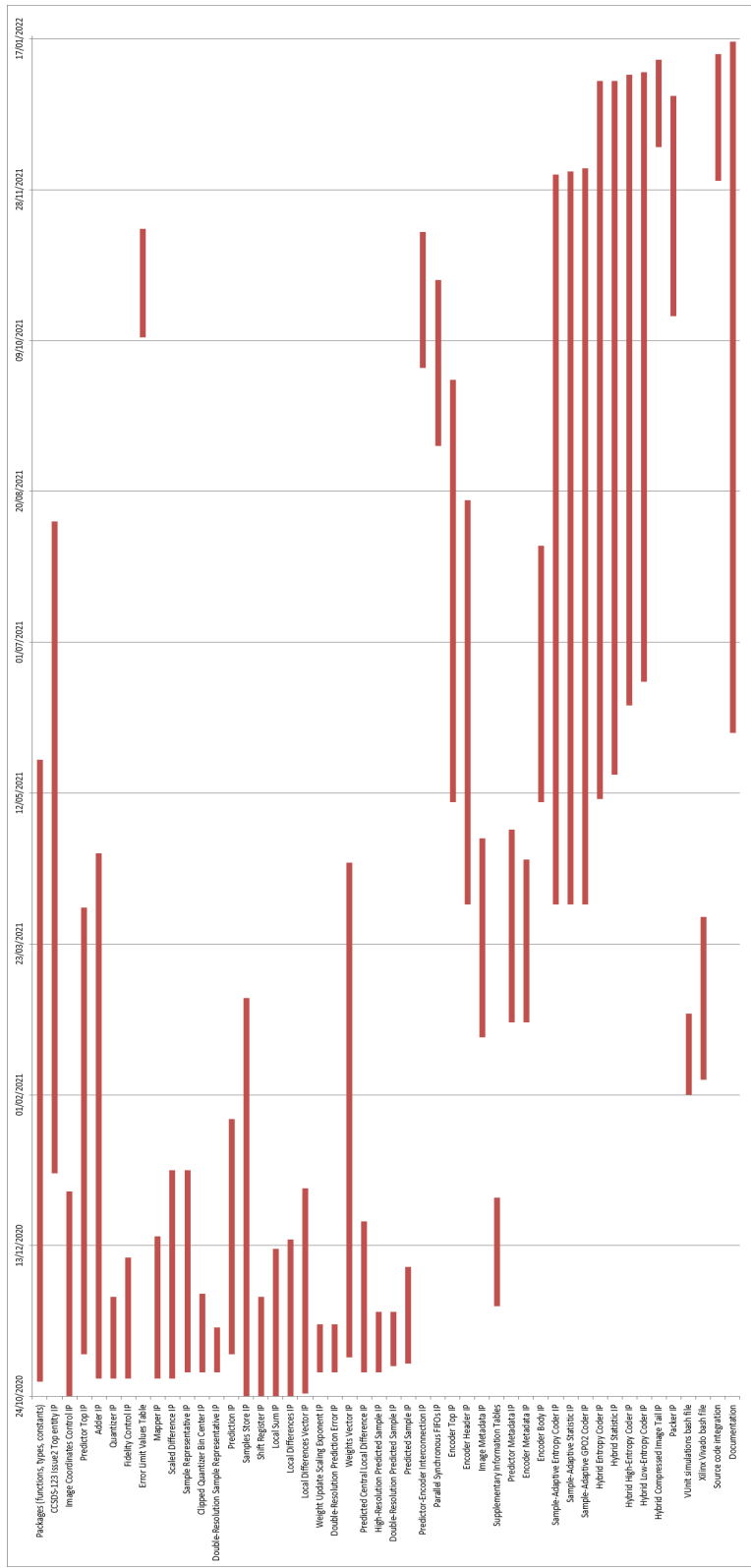


Figure 5.19: Gantt chart

## 6 Discussion

Compared to the previous *Specialization Project* [24], where the basis of the current work comes from, now the *Master Thesis* completes the system, a work with fairly more than 7.5+30 ECTS behind its source code.

The implementation with a modular and reusable approach continues to be the right direction to go one more time. Keeping one IP per (almost) every single mathematical operation (having a 1:1 match with the structure of the provided documentation) has definitely ease the development and the validation of the source code. This could be seen while testing and debugging the system, specially in the two close-loop branches from the *Predictor* block.

And in the same way, the source code is considered to be prepared for future improvements or bug fixes.

Now that the whole system has been implemented, the most complex part of it has been with no doubt the *Encoder* block. While it is true that such block is smaller than the *Predictor* block, its nature of variable-length output signals (for *Encoder Header*, *Encoder Body* and *Packer* parts) required a lot of time for design and simulation, in order to ensure the source code would work under any possible user configuration.

The VUnit and TCL frameworks, commanded by their respective bash files, resulted to be very powerful tools to automate a lot of work, specially when using the *Xilinx Vivado IDE*, a very demanding and quite slow software tool.

For simulation, the defined VUnit testbenches architecture along with the *ModelSim Simulator Engine* were very useful to quickly test and debug the IPs. However, it is also true that the validation process was quite time-consuming because there was no time to implement automatic pass/fail criteria assertions, which could have potentially automated all the source code validation (even though the fully-configurable nature of the source code would have made really difficult this to happen).

Always keeping in mind that the results can slightly change depending on the selected user configuration, the bitstream takes very little time to be generated, around **20 minutes** (since this is a pure PL design), and it demands only around **10% of the resources** from the FPGA target device.

Furthermore, the clock frequency had to be dropped to **40MHz** in order to meet all the timing constraints, instead of the original expectations of around **100-125MHz**. This was because the equations have not been broken down in several steps per clock cycle, yet everything wanted to be done at once.

While this unforeseen issue with the timing constraints could be seen as a design problem, the truth is quite the contrary. Thanks to the aforementioned modular and reusable source code architecture, already created thinking in the improvements and bug fixes coming for the future, it turns out that such fix of breaking down the equations is quite easy to perform, not affecting the integrity of the design at all.

Indeed, this proves one more time how robust the presented source code is, and how well prepared for the future it is.

Last but not least, the implemented source code is functionally working as intended, statement validated from 2 different approaches: visual validation of mathematical operations and *signals* synchronization, and comparing the results with the VHDL implementation from reference [17].

Additionally, equation 5.3 defines the **total number of clock cycles** to fully compress the input image, and an *image compression ratio* of **between 40% and 60%** the original input image size (depending on the selected user configuration, specially by the output packet size) has been measured.

## 7 Related Work

A bunch of papers have been written proposing partial implementations of the CCSDS-123 Issue 1 & 2 algorithms to be run on configurable hardware (FPGA), but so far there are no full implementations of them, which makes the present work be the state-of-the-art of the CCSDS-123 standard.

For example, references [17] and [18] introduce a partial (but completely functional) implementation of the Issue 1 algorithm, tested under many different FPGA platforms and with an outstanding performance.

Reference [9] is also a partial implementation with good performance results, but developed in C, to run on a uC.

Other papers just focus on the *Hybrid Entropy Coder*, the unique new *Entropy Coder* introduced in the Issue 2 algorithm, and also the most complex among the three available ones (together with *Sample-Adaptive Entropy Coder* and *Block-Adaptive Entropy Coder*).

Reference [14] is a study of its insights and capabilities, and references [21] and [26] introduce a complete implementation of the same *Entropy Coder*, detailing the necessary resources to build it up along with its performance results.

Reference [28] introduces a partial implementation of the Issue 2 by using HLS, giving a software approach to the algorithm, but still to be run on configurable hardware. Performance and resources usage are also discussed in here, but being less size-efficient compared to a pure HDL implementation.

Finally, a quite remarkable reference is [25]. It introduces an implementation of the Issue 2 (only the new features in respect of Issue 1), yet the really interesting point is the proposal of using a new samples input order apart from BSQ, BIP and BIL: the *Frame Interleaved by Diagonal (FID)*. This input order promises to reduce the dependencies with the neighbour samples on the *Predictor* stage, and so, to perform a lighter and faster payload compression.

None of these implementations integrates the *Encoder Header* (section 3.9.1), *Supplementary Information Tables* (section 3.9.1.1.1) nor *Error Limit values* update (section 3.7.2.1.1) parts, to say some examples, and this is the reason why the present work is currently by far the most complete implementation of the CCSDS-123 Issue 1 & 2 algorithms, integrating the whole CCSDS-123 standard, with the unique exception of the *Block-Adaptive Entropy Coder* (logic coming from Issue 1).

Nonetheless, the aforementioned works have been excellent references for this one, and hopefully the present report will become a reference point for future papers as well.



## 8 Future Work

The present work is the most complete implementation of the CCSDS-123 Issues 1 & 2 algorithm released as of today, at least for the FPGA world, and providing very positive results so far. Nonetheless, there are still some things to be done in order to make this implementation more robust and to give it a lot of added value.

The very first point is to break down all implemented equations on the design, so that a single mathematical operation per clock cycle is executed. This would let increase the systems' clock frequency from the current 40MHz to around 100-125MHz or even more. Even though that would make the system need more clock cycles to successfully compress an image, in terms of data-rate would be a big improvement.

The interesting part of this point is that is quite easy to perform, as the source code architecture has been developed in a way that this kind of improvements or bug fixes can be done with a minimum effort, not affecting the source code integrity at all.

Another point would be to implement the *Block-Adaptive Entropy Coder*. This IP comes from the Issue 1 algorithm (not updated on Issue 2), and it is the very last remaining piece of logic in here to have a 100% complete implementation of the CCSDS-123 standard. This *Entropy Coder* is less complex than the new *Hybrid Entropy Coder*, so its implementation should not be time-consuming at all.

Next would be the validation by hardware of the source code, as the pandemic situation with Covid-19 did not give any change to work on the assigned hardware platform. And although extensive validation by simulation has been performed, the fully-configurable nature of this algorithm always leaves an open door for creating more *test-cases*, and to provide even more consistent results. The *Predictor* block is the part of the code with more configurable parameters, so it would be smart to start from that section on.

Once improvements have been suggested, the next statements are referred to the added value, and the most interesting one is to integrate a PS-PL communication to let the PS either read or modify the PL configuration in real-time.

Now, all system configuration is static, which means re-compilation is mandatory every time a parameter needs to be modified, but adding an AXI-Lite interface in the very top IP of the design, for example, to receive in real-time the total *image coordinates* of the next image, would prepare this algorithm to (de)code images of different sizes without stop working at all. This proposal opens up a huge range of new possibilities indeed.

All these proposals and suggestions ensure a more reliable and robust solution to be used in the outer space with no problems, and also prepared to overcome the new challenges that are about to come.

## 9 Conclusions

The present report introduces a VHDL-2008 full implementation of the CCSDS-123 Issue 1 & 2 algorithms (with the unique exception of the *Block-Adaptive Entropy Coder*) to run on a FPGA. To the best of the author's knowledge, this work represents the current state-of-the-art and it pretends to be a reference point to all future works related with the CCSDS-123 standard.

The source code has been designed with the *modularity*, *reusability* and *readability* principles as its core. Implementing a single IP for (almost) every single mathematical operation, it offers the possibility to update or fix the code with a minimum effort (so already prepared for the future) as well as a 1:1 match with the provided documentation (so really fast to understand it).

The VUnit framework has been used to perform validation by simulation, which automated and accelerated this process a lot, and all executed tests were successful, so the implementation is already prepared to work. Anyway, because of the fully-configurable nature of this implementation, it was not possible to test every possible *test-case*.

Besides, even though the pandemic situation with the Covid-19 did not give any chance to perform validation by hardware, a TCL framework is also provided to automate the creation of a *Xilinx Vivado* project along with the bitstream and HW integration reports generation (taking about 20 minutes to finish). This framework saves a lot of time while developing as well.

The complete system just uses a single clock signal to work, and the timing report shows that a maximum clock frequency of 40MHz can be used to perform all operations on time. Breaking down all implemented equations, to execute one mathematical operation per clock cycle, something really easy to do thanks to the introduced modular and reusable source code architecture, would increase such clock frequency to around 100-125MHz, or even more.

Furthermore, this implementation only requires around 10% of the resources from the chosen target device: *Zynq UltraScale+ MPSoC ZCU102 Evaluation Board*. Nevertheless, the results can slightly change depending on the selected user configuration.

The total clock cycles require to fully compress an input image is precisely defined on equation 5.3, very closely related to such image dimensions, showing that the Issue 2 requires substantially more time to finish than Issue 1 because of the two new close-loop branches.

Besides, an *image compression ratio* of between 40% and 60% in respect of the original input image size, depending on the user configuration, has been measured.

Thanks to the aforementioned design principles applied while developing the code, the expectations now are that this implementation is used on a real environment for the (de)compression of hyperspectral images under the CCSDS-123 standard, and in case the source code shall be either upgraded or fixed, a new solution should not be developed, but instead, this work should act as a basis for the improvements, with the aim of creating a very robust commercial solution of the CCSDS-123 standard to run on FPGA.

Hopefully this work can be a good contribution to the HYPSONO mission and SmallSat project in the NTNU, and it can help to many others projects in the future.

# 10 Appendix - List of codes

## 10.1 Mathematical conventions

In order to interpret all equations from chapter 2 properly, the following mathematical notations must be understood first:

- The largest integer  $n$  such that  $n \leq x$ :

$$n = \lfloor x \rfloor \quad (10.1)$$

- The smallest integer  $n$  such that  $n \geq x$ :

$$n = \lceil x \rceil \quad (10.2)$$

- The modulus of an integer  $M$  with respect to a positive integer divisor  $n$ :

$$M \bmod n = M - n \lfloor M / n \rfloor \quad (10.3)$$

- The  $R$ -bit two's complement integer that is congruent to  $x$  modulo  $2^R$ :

$$\text{mod}_R^*[x] = \left( (x + 2^{R-1}) \bmod 2^R \right) - 2^{R-1} \quad (10.4)$$

- The clipping of the real number  $x$  to the range  $[x_{\min}, x_{\max}]$ :

$$\text{clip}(x, \{x_{\min}, x_{\max}\}) = \begin{cases} x_{\min}, & x < x_{\min} \\ x, & x_{\min} \leq x \leq x_{\max} \\ x_{\max}, & x > x_{\max} \end{cases} \quad (10.5)$$

- The sign function  $\text{sgn}(x)$ :

$$\text{sgn}(x) = \begin{cases} 1, & x > 0 \\ 0, & x = 0 \\ -1, & x < 0 \end{cases} \quad (10.6)$$

- The sign-plus function  $\text{sgn}^+(x)$ :

$$\text{sgn}^+(x) = \begin{cases} 1, & x \geq 0 \\ -1, & x < 0 \end{cases} \quad (10.7)$$

## 10.2 Image parameters

Symbol	Meaning
$N_X, N_Y, N_Z$	image dimensions (user-specified)
$D$	image dynamic range in bits (user-specified)
$S_{\min}, S_{\max}$	lower and upper sample value limits
$S_{\text{mid}}$	mid-range sample value
$\tau$	number of supplementary information tables (user-specified)
$D_I$	integer supplementary information table bit depth (optional, user-specified)
$\beta$	float supplementary information table exponent bias (optional, user-specified)
$D_F$	float supplementary information table significand bit depth (optional, user-specified)
$D_E$	float supplementary information table exponent bit depth (optional, user-specified)

Table 10.1: Image parameters [10, p.98]

## 10.3 Predictor parameters

Symbol	Meaning
$P$	number of spectral bands used for prediction (user-specified)
$P_z^*$	number of previous spectral bands used for prediction in band $z$
$C_z$	number of local difference values used for prediction in band $z$
$\Omega$	weight resolution (user-specified)
$\omega_{\min}, \omega_{\max}$	minimum and maximum weight values
$\Lambda_z$	weight initialization vector (optional, user-specified)
$Q$	weight initialization resolution (optional, user-specified)
$R$	register size, in bits, used in prediction calculation (user-specified)
$a_z$	absolute error limit (optional, user-specified)
$r_z$	relative error limit (optional, user-specified)
$D_A$	absolute error limit bit depth (optional, user-specified)
$D_R$	relative error limit bit depth (optional, user-specified)
$A^*$	absolute error limit constant (optional, user-specified)

Table 10.2: Predictor parameters (1/2) [10, p.98-100]

Symbol	Meaning
$R^*$	relative error limit constant (optional, user-specified)
$u$	error limit update period exponent (optional, user-specified)
$\Theta$	sample representative resolution (user-specified)
$\phi_z$	sample representative damping (user-specified)
$\psi_z$	sample representative offset (user-specified)
$v_{\min}, v_{\max}$	initial and final weight update scaling exponent parameters (user-specified)
$t_{\text{inc}}$	weight update scaling exponent change interval (user-specified)
$\zeta_z^{(i)}, \zeta_z^*$	weight update scaling exponent offsets (user-specified)

Table 10.3: Predictor parameters (2/2) [10, p.98-100]

## 10.4 Encoder parameters

Symbol	Meaning
$B$	output word size in bytes (user-specified)
$M$	sub-frame interleaving depth (optional, user-specified)
<b>Sample-Adaptive Entropy Coder</b>	
$U_{\max}$	unary length limit (user-specified)
$\gamma_0$	initial count exponent (user-specified)
$K$	accumulator initialization constant (optional, user-specified)
$\gamma^*$	rescaling counter size (user-specified)
<b>Hybrid Entropy Coder</b>	
$U_{\max}$	unary length limit (user-specified)
$T_i$	low-entropy code threshold value
$L_i$	low-entropy code input symbol limit
$\gamma_0$	initial count exponent (user-specified)
$\gamma^*$	rescaling counter size (user-specified)
<b>Block-Adaptive Entropy Coder</b>	
$n$	resolution
$J$	block size (user-specified)
$r$	reference sample interval (user-specified)

Table 10.4: Encoder parameters [10, p.100-101]

## 10.5 VHDL Style Guide

All source code has been developed as plain-text using only VHDL-2008 (so without high-level tools), and following quite strict coding rules in order to enhance the readability as much as possible.

Hereafter is the complete style guide used for the development [16]:

- Generics and constants are written in upper case (e.g. 'ONE\_CONSTANT') to emphasize them.
- Everything else is written in lower case, including entity/signal names, keywords...
- All identifiers written in C style (e.g. 'long\_identifier\_made\_using\_underscores').
- Components are used to instantiate VHDL entities instead of direct instantiation.
- Component declarations are put in a package if used in more than one place.
- Prefixes and suffixes are used for different identifiers:
  - Non-interface ports:
    - \* In ports: `_i`
    - \* Out ports: `_o`
    - \* I/O ports: `_io`
  - Interface ports (AXI4-Full, AXI4-Stream...):
    - \* Master ("client", making requests): `m_`
    - \* Slave ("server", responding to requests): `s_`
  - Signals: `_s`
  - Variables: `_v`
  - Alias: `_a`
  - Types: `_t`
  - Constants: `_C`
  - Generics: `_G`
  - Generate label: `g_`
  - Process label: `p_`
  - Block label: `b_`
  - Instantiation label: `i_`

## 10.6 HDL Coding Guidelines

The following list shows the guidelines used in this project (and specially for further improvements) to get an easily maintainable source code [16]:

- Write readable, easily understandable, self-explaining code:
  - Take good care about identifier names. Do not use unnecessary abbreviations which make the code hard to read. Make every identifier explain itself and its intention. Take your time to think about proper names.
  - Use functions, procedures and auxiliary variables with strong names to make the code more readable and self-explaining.
  - Use comments to explain the big picture and give background information beyond what is written in the code (e.g. explaining the idea behind a certain algorithm).
  - If you feel the need to write comments explaining what you are doing (or how you are doing it), rewrite the code to explain itself. Comments get outdated easily. The code in contrast is always up to date.
- Do not repeat yourself:
  - Do not copy and paste code. Extract re-usable code into entities, functions or procedures.
  - Do not use "magic numbers" in the code. Use constants, functions or procedures defined at proper places instead (within the smallest scope at which you need them).
  - Make use of the single source of truth: No information or algorithm may be redundant. Everything should be defined in one place only.
- Write modular code:
  - Every module (a VHDL entity, a function, a procedure or a process) should have exactly one job. If a module does several things at once, try to split the module into several smaller ones.
  - Reduce the dependencies between all modules. Each module should have a clear, well-defined interface with minimal functional dependencies (whether explicit or implicit) with other modules. Each module should be able to work on its own.
  - Do not use global constants to configure the code. Use generics instead wherever possible.
  - Use packages to define constants, functions and procedures. Remember that also a package should serve exactly one purpose. Do not randomly assemble unrelated constants, functions and procedures into random package. Everything within a package should be closely related.



- Write reusable code:
  - Whenever you have the chance, break down each problem into smaller and reusable pieces.
  - Use already existing modules wherever possible.
  - Write generic, reusable modules when no modules exist for your problem yet.
  - Communicate! Whenever writing new (possibly) reusable modules, agree on the interface and behaviour with the team.
- Stay at one abstraction layer:
  - Break down your problem top down. Start by writing abstract code at the highest possible level and use black-box modules to perform the more detailed work. Implement these black-box modules using the same approach.
  - Always work as near to the "problem domain" as possible instead of at the "solution domain". Use high level data types (records) which represent the solution to your problem instead of using low level data types (std\_logic) for everything.
  - Do not mix high level and low level code. Do not instantiate FPGA primitives within a high level code. Each code should only depend on your problem (problem domain) but NOT on the hardware OR the other way around. Never write problem specific code which depends on specific hardware.
  - Avoid mixing the level of abstraction within one module. Again, use functions, procedures and entities as interfaces to lower or higher abstraction layers.
- Write automated test-benches for your modules:
  - Think about all major possible use cases of your modules (not only the use cases you are handling right now).
  - Define the behaviour of the modules as generic as possible (of course depending on the nature and the general reusability). Define test cases covering all the use cases.
  - Implement a VHDL test-bench for all the test cases. Each test-bench should have automatic pass/fail checks.
  - Extend the test benches regularly. Whenever implementing new features or fixing bugs, start by writing new test cases for it.
  - Perform regression tests with every new change by running all previously existing test cases.

## 10.7 SoC Package Pinout

Figure 10.1 is the pinout of the SoC XCZU9EG, package FFVB1156 [36, p.145]:

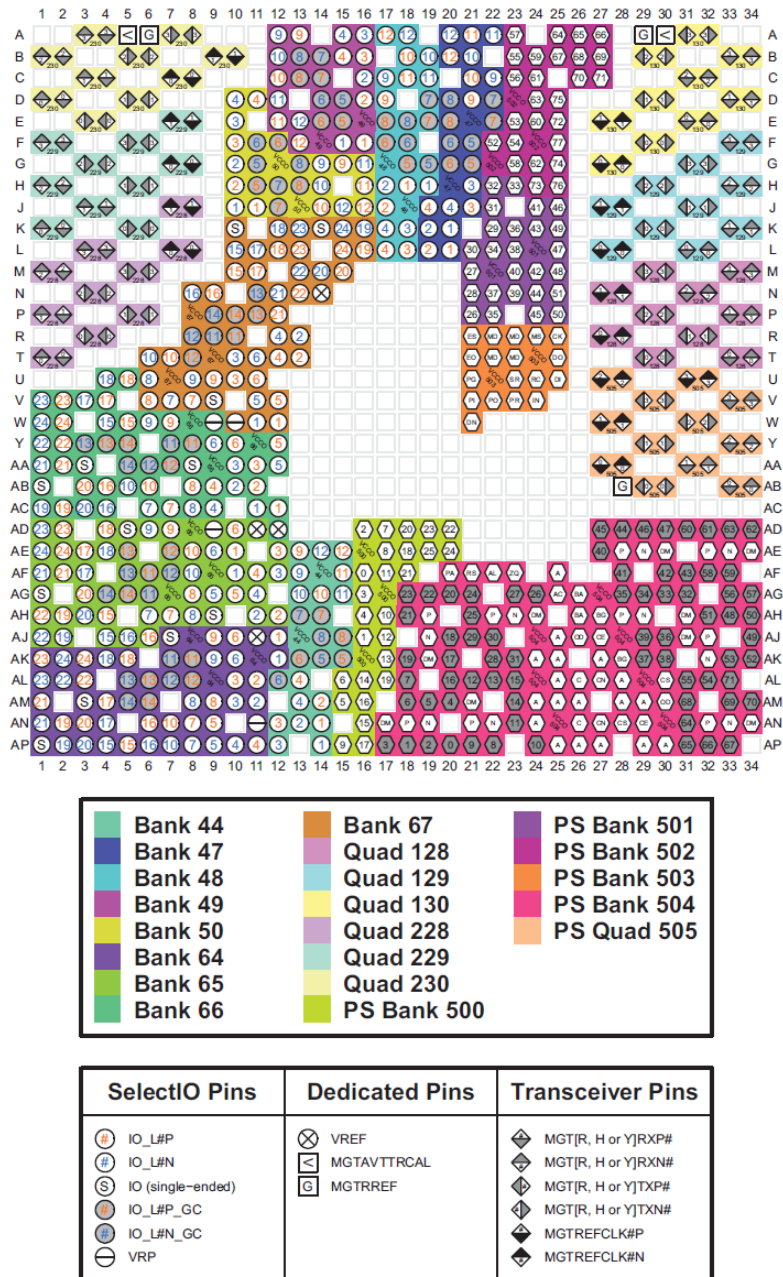


Figure 10.1: SoC XCZU9EG-FFVB1156 package pinout



## 10.9 Signed/Unsigned signals handling

```
1  -- Example 1 for handling "signed" and "unsigned" signals with different data sizes
2  p_example1 : process(clock_i) is
3  begin
4      if rising_edge(clock_i) then
5          if (SMPL_TYPE_G = '0') then -- Incoming data is recognized as "signed"
6              comp1_s <= std_logic_sector(resize(resize(signed(data_s0_i), D_C+3) -
7                  resize(signed(data_s3_i), D_C+3), D_C+2));
8          else -- SMPL_TYPE_G = '1' -- Incoming data is recognized as "unsigned"
9              comp1_s <= std_logic_sector(resize(resize(unsigned(data_s0_i), D_C+3))
10                 - resize(signed(data_s3_i), D_C+3), D_C+2));
11          end if;
12      end process p_example1;
13
14  -- Example 2 for handling "signed" and "unsigned" signals with different data sizes
15  p_example2 : process(clock_i) is
16  begin
17      if rising_edge(clock_i) then
18          comp1_s <= std_logic_sector(to_signed(4 - fi_s, comp1_s'length));
19          comp2_s <= std_logic_sector(resize(shift_sgn(resize(signed(data_s1_i), Re_C),
20              OMEGA_C), Re_C));
21          comp3_s <= sgn(signed(data_quant_i));
22          comp4_s <= std_logic_sector(resize(to_signed(comp3_s, Re_C*2) *
23              resize(signed(data_merr_i), Re_C*2), Re_C*2));
24          comp5_s <= std_logic_sector(resize(signed(comp2_s) - signed(comp4_s), Re_C*2));
25          comp6_s <= std_logic_sector(resize(resize(signed(data_s6_i), Re_C) -
26              resize(signed(PW_OM1_C), Re_C), Re_C));
27          comp7_s <= std_logic_sector(resize(resize(signed(comp1_s), Re_C*2) *
28              resize(signed(comp5_s), Re_C*2) + resize(signed(comp6_s), Re_C*2), Re_C*2));
29          comp8_s <= std_logic_sector(resize(shift_sgn(signed(comp7_s), -D_C), Re_C));
30      end if;
31  end process p_example2;
32
33  -- Example 3 for handling "signed" and "unsigned" signals with different data sizes
34  p_example3 : process(clock_i) is
35  begin
36      if rising_edge(clock_i) then
37          comp1_s <= std_logic_sector(resize(resize(signed(data_lsum_i), D_C+6) -
38              shift_sgn(resize(signed(SMPL_LIMIT_G.mid), D_C+6), 2), D_C+6));
39          comp2_s <= std_logic_sector(resize(resize(signed(data_pred_cldiff_i), Re_C+4) +
40              shift_sgn(resize(signed(comp1_s), Re_C+4), OMEGA_C), Re_C+4));
41          comp3_s <= std_logic_sector(mod_R(signed(comp2_s), Re_C));
42          comp4_s <= std_logic_sector(resize(resize(signed(comp3_s), Re_C) +
43              shift_sgn(resize(signed(SMPL_LIMIT_G.mid), Re_C), OMEGA_C+2) +
44              resize(signed(PW_OM1_C), Re_C), Re_C));
45          comp5_s <= std_logic_sector(resize(shift_sgn(resize(signed(SMPL_LIMIT_G.min),
46              Re_C), OMEGA_C+2), Re_C));
47          comp6_s <= std_logic_sector(resize(shift_sgn(resize(signed(SMPL_LIMIT_G.max),
48              Re_C), OMEGA_C+2) + resize(signed(PW_OM1_C), Re_C), Re_C));
49          comp7_s <= std_logic_sector(clip(signed(comp4_s), signed(comp5_s),
50              signed(comp6_s)));
51      end if;
52  end process p_example3;
```

## 10.10 VHDL Package example

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  library work;
6  use work.param_image.all;
7  use work.types_image.all;
8  use work.utils_image.all;
9
10 -- Table E-2: Predictor Quantities
11 package param_predictor is
12
13     constant P_C      : integer range 0 to 15          := work.utils_image.min(3,
NZ_C); -- Number of spectral bands used for prediction
14     constant MAX_PZ_C: integer range 0 to 15          := P_C;
-- Maximum number of previous spectral bands for prediction in band Z
15     constant MAX_CZ_C: integer range 0 to MAX_PZ_C+3  := P_C + 3;
-- Maximum number of local diff. values for prediction in band Z
16
17     constant DA_C     : integer range 1 to (work.utils_image.min(D_C-1, 16)) := 8;
-- Absolute error limit bit depth
18     constant A_C     : integer range 0 to (2**DA_C-1)  := 10;
-- Absolute error limit constant
19     constant DR_C     : integer range 1 to (work.utils_image.min(D_C-1, 16)) := 8;
-- Relative error limit bit depth
20     constant R_C     : integer range 0 to (2**DR_C-1)  := 20;
-- Relative error limit constant
21
22     constant U_C     : integer range 0 to 9            := 2;
-- Error limit update period exponent
23
24     constant THETA_C : integer range 0 to 4           := 0;
-- Sample representative resolution
25     constant FI_ARR_C : array_integer_t(0 to NZ_C-1) := (others => 0);
-- Sample representative damping
26     constant PSI_ARR_C: array_integer_t(0 to NZ_C-1) := (others => 0);
-- Sample representative offset
27
28     constant C_ARR_C : array_integer_t(0 to NZ_C-1) := (others => 0);
-- Intra-band weight exponent offsets
29     constant Ci_MTX_C: matrix_integer_t(0 to NZ_C-1)(0 to MAX_PZ_C-1) := (others =>
(others => 0)); -- Inter-band weight exponent offsets
30
31     constant V_MIN_C : integer range -6 to 9          := -6;
-- Initial weight update scaling exponent parameters
32     constant V_MAX_C : integer range V_MIN_C to 9     := 9;
-- Final weight update scaling exponent parameters
33     constant T_INC_C : integer range 4 to 11          := 4;
-- Weight update scaling exponent change interval
34
35     constant OMEGA_C : integer range 4 to 19          := 19;
-- Weight resolution
36     constant W_MIN_C : signed(OMEGA_C+3-1 downto 0)  := (OMEGA_C+3-1 => '1',
others => '0'); -- Minimum possible weight value
37     constant W_MAX_C : signed(OMEGA_C+3-1 downto 0)  := (OMEGA_C+3-1 => '0',
others => '1'); -- Maximum possible weight value
38     constant Q_C     : integer range 3 to (OMEGA_C+3) := 8;
-- Weight initialization resolution
39     constant Re_C    : integer range (work.utils_image.max(32, D_C+OMEGA_C+2)) to 64
:= 64; -- Register size in bits, used in prediction calculation
40
41     -- Weight initialization vector (in two's complement)
42     constant LAMBDA_MTX_C : matrix_signed_t(0 to NZ_C-1)(0 to MAX_CZ_C-1)(Q_C-1
downto 0) := (others => (others => (Q_C/2-1 => '1', others => '0')));
43
44 end package param_predictor;

```

## 10.11 Image Coordinates Control IP source code

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  library work;
6  use work.gen_labels.all;
7  use work.param_image.all;
8  use work.types_image.all;
9  use work.utils_image.all;
10
11 use work.param_encoder.all;
12
13 entity image_coord_control is
14   generic (
15     -- 00: BSQ order, 01: BIP order, 10: BIL order
16     SMPL_ORDER_G : std_logic_vector(1 downto 0)
17   );
18   port (
19     clock_i      : in  std_logic;
20     reset_i      : in  std_logic;
21
22     enable_i     : in  std_logic;
23     enable_o     : out std_logic;
24
25     image_end_i  : in  std_logic;
26     img_coord_o  : out img_coord_t
27   );
28 end entity image_coord_control;
29
30 architecture behavioural of image_coord_control is
31   -- Regardless of the sample order, ONLY the first coordinate to change must
32   -- start with '-1' (either 'x' or 'z'), so that the first change outputs (0, 0, 0)
33   signal x_s : integer range -1 to NX_C-1 := iif(SMPL_ORDER_G = BIP_C, 0, -1);
34   -- X coord: Sample/Width
35   signal y_s : integer range -1 to NY_C-1 := 0;
36   -- Y coord: Frame
37   signal z_s : integer range -1 to NZ_C-1 := iif(SMPL_ORDER_G = BIP_C, -1, 0);
38   -- Z coord: (Spectral) Band
39
40   signal enable_s      : std_logic := '0';
41   signal image_end_s   : std_logic := '0';
42   signal stall_s      : std_logic := '0';
43
44   -- Intermediate values (only for BIP/BIL input order)
45   constant M_C        : integer := iif(SMPL_ORDER_G = BIP_C, M_BIP_C, M_BIL_C);
46   constant I_MAX_C    : integer := division_up(NZ_C, M_C);
47
48 begin
49   -- Input values delayed to synchronize them with the next modules in chain
50   p_image_delay : process(clock_i) is
51   begin
52     if rising_edge(clock_i) then
53       if (reset_i = '1') then
54         image_end_s <= '0';
55         enable_s    <= '0';
56       else
57         -- "image_end_s" is the only signal that must not be restarted with
58         -- "stall_s", so such condition only applies for "enable_s"
59         image_end_s <= image_end_i;
60         if (stall_s = '1') then
61           enable_s <= '0';
62         else
63           enable_s <= enable_i;
64         end if;
65       end if;
66     end if;
67   end process p_image_delay;
68
69   146
```

```

65     -- Once whole image has been introduced, next one must not start until current
66     one has been completely processed
67     p_stall_count : process(reset_i, image_end_s, image_end_i, x_s, y_s, z_s) is
68     begin
69         if (reset_i = '1') then
70             stall_s <= '0';
71         else
72             if (x_s >= NX_C-1 and y_s >= NY_C-1 and z_s >= NZ_C-1) then
73                 stall_s <= '1';
74             else
75                 -- Due to synchro. issues, the deassertion of "stall_s" can ONLY be
76                 done with a falling edge of "image_end_i"
77                 if (image_end_s = '1' and image_end_i = '0') then
78                     stall_s <= '0';
79                 end if;
80             end if;
81         end process p_stall_count;
82
83     -- Coordinates counting for input order BSQ
84     g_img_coord_BSQ : if (SMPL_ORDER_G = BSQ_C) generate
85         p_img_coord_BSQ : process(clock_i) is
86         begin
87             if (rising_edge(clock_i)) then
88                 if (reset_i = '1' or stall_s = '1') then
89                     x_s <= -1;      -- With BSQ_C, first coordinate to change is
90                     always 'x', so it should always start with '-1'
91                     y_s <= 0;
92                     z_s <= 0;
93                 else
94                     if (enable_i = '1') then
95                         if (x_s < NX_C-1) then
96                             x_s <= x_s + 1;
97                         else
98                             x_s <= 0;
99                             if (y_s < NY_C-1) then
100                                 y_s <= y_s + 1;
101                             else
102                                 y_s <= 0;
103                                 if (z_s < NZ_C-1) then
104                                     z_s <= z_s + 1;
105                                 else
106                                     z_s <= 0;
107                                 end if;
108                             end if;
109                         end if;
110                     end if;
111                 end process p_img_coord_BSQ;
112             end generate g_img_coord_BSQ;
113
114     -- Outputs
115     enable_o    <= enable_s;
116     img_coord_o <= (
117         x    => x_s,
118         y    => y_s,
119         z    => z_s,
120         t    => y_s * NX_C + x_s
121     );
122 end architecture behavioural;

```

## 10.12 Adder IP source code

```
1  entity adder is
2    generic (
3      SMPL_TYPE_G      : std_logic;
4      SMPL_ORDER_G    : std_logic_vector(1 downto 0)
5    );
6    port (
7      clock_i          : in  std_logic;
8      reset_i          : in  std_logic;
9
10     img_en_i         : in  std_logic;
11     enable_o         : out std_logic;
12     img_coord_i      : in  img_coord_t;
13     img_coord_o      : out img_coord_t;
14
15     pred_en_i        : in  std_logic;
16     pred_coord_i     : in  img_coord_t;
17     data_s3_i        : in  std_logic_vector(D_C+1-1 downto 0);
18     data_s3_o        : out std_logic_vector(D_C+1-1 downto 0);
19     data_s4_lsb_i    : in  std_logic;
20     data_s4_lsb_o    : out std_logic;
21
22     data_s0_i        : in  std_logic_vector(D_C-1 downto 0);
23     en_res_o         : out std_logic;
24     data_res_o       : out std_logic_vector(D_C+2-1 downto 0)
25   );
26 end entity adder;
27
28 architecture behavioural of adder is
29 begin
30   p_cnt_flags : process(clock_i) is
31   begin
32     if rising_edge(clock_i) then
33       if (reset_i = '1') then
34         img_cnt_s      <= 0;
35         pred_cnt_s     <= 0;
36         final_cnt_s    <= 0;
37         img_end_flag_s <= '0';
38         pred_end_flag_s <= '0';
39         data_s4_lsb_arr_s <= (others => '0');
40         data_s0_arr_s   <= (others => (others => '0'));
41         data_s3_arr_s   <= (others => (others => '0'));
42         img_coord_arr_s <= (others => reset_img_coord(SMPL_ORDER_G));
43       else
44         -- Monitors and saves incoming original input samples
45         if (img_en_i = '1' and img_end_flag_s = '0') then
46           data_s0_arr_s(img_cnt_s) <= data_s0_i;
47           img_coord_arr_s(img_cnt_s) <= img_coord_i;
48           if (img_cnt_s < IMG_SIZE_C-1) then
49             img_cnt_s <= img_cnt_s + 1;
50           else
51             img_cnt_s <= 0;
52             img_end_flag_s <= '1';
53           end if;
54         end if;
55       end if;
56
57       -- Monitors and saves incoming predicted samples
58       if (pred_en_i = '1' and pred_end_flag_s = '0') then
59         data_s3_arr_s(pred_cnt_s) <= data_s3_i;
60         if (pred_cnt_s < IMG_SIZE_C-1) then
61           pred_cnt_s <= pred_cnt_s + 1;
62         else
63           pred_cnt_s <= 0;
64           pred_end_flag_s <= '1';
65         end if;
66         data_s4_lsb_arr_s(pred_cnt_s) <= data_s4_lsb_i;
67       end if;
68
69       -- Monitors and waits until residual values are sent out
70       if (img_end_flag_s = '1' and pred_end_flag_s = '1') then
```



```

70         if (final_cnt_s < IMG_SIZE_C-1) then
71             final_cnt_s <= final_cnt_s + 1;
72         else
73             final_cnt_s <= 0;
74             img_end_flag_s <= '0';
75             pred_end_flag_s <= '0';
76         end if;
77     end if;
78 end if;
79 end if;
80 end process p_cnt_flags;
81
82 p_pred_res_calc : process(clock_i) is
83 begin
84     if rising_edge(clock_i) then
85         if (reset_i = '1') then
86             data_s3_s <= (others => '0');
87             data_s4_lsb_s <= '0';
88             en_res_s <= '0';
89             data_res_s <= (others => '0');
90         else
91             if (img_en_i = '1' or pred_en_i = '1' or (img_end_flag_s = '1' and
92 pred_end_flag_s = '1')) then
93                 if (img_end_flag_s = '0' or pred_end_flag_s = '0') then
94                     if (img_end_flag_s = '0') then
95                         if (SMPL_TYPE_G = '0') then
96                             data_res_s <=
97                                 std_logic_vector(resize(resize(signed(data_s0_i),
98 D_C+3) - resize(signed(data_s3_i), D_C+3), D_C+2));
99                         else -- SMPL_TYPE_G = '1'
100                             data_res_s <=
101                                 std_logic_vector(resize(signed(resize(unsigned(data_s0
102 _i), D_C+3)) - resize(signed(data_s3_i), D_C+3),
103 D_C+2));
104                         end if;
105                     end if;
106                     if (pred_end_flag_s = '0') then
107                         data_s3_s <= data_s3_i;
108                     end if;
109                 elsif (img_end_flag_s = '1' and pred_end_flag_s = '1') then
110                     data_s3_s <= data_s3_arr_s(final_cnt_s);
111                     data_s4_lsb_s <= data_s4_lsb_arr_s(final_cnt_s);
112                     en_res_s <= '1';
113                     if (SMPL_TYPE_G = '0') then
114                         data_res_s <=
115                             std_logic_vector(resize(resize(signed(data_s0_arr_s(final_
116 cnt_s)), D_C+3) -
117 resize(signed(data_s3_arr_s(final_cnt_s)), D_C+3),
118 D_C+2));
119                     else -- SMPL_TYPE_G = '1'
120                         data_res_s <=
121                             std_logic_vector(resize(signed(resize(unsigned(data_s0_arr
122 _s(final_cnt_s)), D_C+3)) -
123 resize(signed(data_s3_arr_s(final_cnt_s)), D_C+3),
124 D_C+2));
125                     end if;
126                 end if;
127             end if;
128         end if;
129     end if;
130 end process p_pred_res_calc;
131
132 enable_o <= enable_s;
133 img_coord_o <= img_coord_s;
134 data_s3_o <= data_s3_s;
135 data_s4_lsb_o <= data_s4_lsb_s;
136 en_res_o <= en_res_s;
137 data_res_o <= data_res_s;
138 end architecture behavioural;

```

## 10.13 Scaled Difference IP source code

```

1  entity scaled_diff is
2      generic (
3          SMPL_ORDER_G      : std_logic_vector(1 downto 0);
4          SMPL_LIMIT_G      : smpl_lim_t
5      );
6      port (
7          clock_i           : in  std_logic;
8          reset_i           : in  std_logic;
9          enable_i          : in  std_logic;
10         enable_o          : out std_logic;
11         img_coord_i       : in  img_coord_t;
12         img_coord_o       : out img_coord_t;
13         data_s3_i         : in  std_logic_vector(D_C+1-1 downto 0);
14         data_merr_i       : in  std_logic_vector(D_C+1-1 downto 0);
15         scaled_diff_o     : out std_logic_vector(D_C+1-1 downto 0)
16     );
17 end entity scaled_diff;
18
19 architecture behavioural of scaled_diff is
20 begin
21     p_sc_diff_calc : process(clock_i) is
22         variable comp1_v, comp2_v, comp3_v, comp4_v, comp5_v :
23             std_logic_vector(D_C+3-1 downto 0) := (others => '0');
24     begin
25         if rising_edge(clock_i) then
26             if (reset_i = '1') then
27                 comp1_v := (others => '0');
28                 comp2_v := (others => '0');
29                 comp3_v := (others => '0');
30                 comp4_v := (others => '0');
31                 comp5_v := (others => '0');
32                 scaled_diff_s <= (others => '0');
33             else
34                 if (enable_i = '1') then
35                     if (img_coord_i.t = 0) then
36                         comp1_v := std_logic_vector(resize(resize(signed(data_s3_i),
37                             D_C+3) - resize(signed(SMPL_LIMIT_G.min), D_C+3), D_C+3));
38                         comp2_v :=
39                             std_logic_vector(resize(resize(signed(SMPL_LIMIT_G.max),
40                                 D_C+3) - resize(signed(data_s3_i), D_C+3), D_C+3));
41                         scaled_diff_s <=
42                             std_logic_vector(resize(work.utils_image.min(signed(comp1_v),
43                                 signed(comp2_v)), D_C+1));
44                     else -- img_coord_i.t > 0
45                         comp3_v := std_logic_vector(resize(resize(signed(data_s3_i),
46                             D_C+3) - resize(signed(SMPL_LIMIT_G.min), D_C+3) +
47                             resize(signed(data_merr_i), D_C+3), D_C+3));
48                         comp4_v :=
49                             std_logic_vector(resize(resize(signed(SMPL_LIMIT_G.max),
50                                 D_C+3) - resize(signed(data_s3_i), D_C+3) +
51                             resize(signed(data_merr_i), D_C+3), D_C+3));
52                         comp5_v :=
53                             std_logic_vector(resize(shift_sgn(resize(signed(data_merr_i),
54                                 D_C+4), 1) + to_signed(1, D_C+4), D_C+3));
55                         scaled_diff_s <=
56                             std_logic_vector(resize(work.utils_image.min(division_down(sig
57                                 ned(comp3_v), signed(comp5_v)),
58                                 division_down(signed(comp4_v), signed(comp5_v))), D_C+1));
59                     end if;
60                 end if;
61             end if;
62         end if;
63     end process p_sc_diff_calc;
64
65     enable_o      <= enable_s;
66     img_coord_o  <= img_coord_s;
67     scaled_diff_o <= scaled_diff_s;
68 end architecture behavioural;

```

## 10.14 Shift Register IP source code

```
1  entity shift_register is
2      generic (
3          SMPL_ORDER_G : std_logic_vector(1 downto 0);
4          DATA_SIZE_G : integer;
5          REG_SIZE_G   : integer;
6          INIT_VAL_G   : std_logic_vector(DATA_SIZE_G-1 downto 0)
7      );
8      port (
9          clock_i      : in  std_logic;
10         reset_i       : in  std_logic;
11
12         enable_i      : in  std_logic;
13         enable_o      : out std_logic;
14         img_coord_i   : in  img_coord_t;
15         img_coord_o   : out img_coord_t;
16
17         data_i        : in  std_logic_vector(DATA_SIZE_G-1 downto 0);
18         data_o        : out std_logic_vector(DATA_SIZE_G-1 downto 0)
19     );
20 end entity shift_register;
21
22 architecture behavioural of shift_register is
23 begin
24     p_shift_reg_array : process(clock_i) is
25     begin
26         if (rising_edge(clock_i)) then
27             if (reset_i = '1') then
28                 enable_s      <= '0';
29                 img_coord_s   <= reset_img_coord(SMPL_ORDER_G);
30                 data_s        <= INIT_VAL_G;
31                 counter_s     <= 0;
32                 enable_arr_s  <= (others => '0');
33                 img_coord_arr_s <= (others => reset_img_coord(SMPL_ORDER_G));
34                 data_arr_s    <= (others => INIT_VAL_G);
35             else
36                 if (REG_SIZE_G > 0) then
37                     enable_arr_s(enable_arr_s'high downto 1) <=
38                     enable_arr_s(enable_arr_s'high-1 downto 0);
39                     img_coord_arr_s(1 to img_coord_arr_s'high) <= img_coord_arr_s(0
40                     to img_coord_arr_s'high-1);
41                     data_arr_s(1 to data_arr_s'high) <= data_arr_s(0 to
42                     data_arr_s'high-1);
43                     enable_arr_s(0) <= enable_i;
44                     img_coord_arr_s(0) <= img_coord_i;
45                     data_arr_s(0) <= data_i;
46                     -- Outcoming value is the highest position from the shift register
47                     enable_s <= enable_arr_s(enable_arr_s'high);
48                     img_coord_s <= img_coord_arr_s(img_coord_arr_s'high);
49                     data_s <= data_arr_s(data_arr_s'high);
50                 else
51                     -- If delay is just 1 clock cycle, input is directly outputted
52                     enable_s <= enable_i;
53                     img_coord_s <= img_coord_i;
54                     data_s <= data_i;
55                 end if;
56             end if;
57
58             if (counter_s < REG_SIZE_G) then
59                 counter_s <= counter_s + 1;
60             else
61                 counter_s <= 0;
62             end if;
63         end if;
64     end process p_shift_reg_array;
65
66     enable_o <= enable_s;
67     img_coord_o <= img_coord_s;
68     data_o <= data_s;
69 end architecture behavioural;
```

## 10.15 Image Metadata IP source code

```
1  entity metadata_img is
2      generic (
3          -- 0: "signed" type samples, 1: "unsigned" type samples
4          SMPL_TYPE_G          : std_logic;
5          -- 00: BSQ order, 01: BIP order, 10: BIL order
6          SMPL_ORDER_G        : std_logic_vector(1 downto 0);
7          -- 00: lossless, 01: absolute error limit only, 10: relative error limit
           only, 11: both absolute and relative error limits
8          FIDEL_CTRL_TYPE_G   : std_logic_vector(1 downto 0);
9          -- 00: Sample-Adaptive Entropy, 01: Hybrid Entropy, 10: Block-Adaptive Entropy
10         ENCODER_TYPE_G      : std_logic_vector(1 downto 0);
11         -- User Defined Data
12         UDEF_DATA_G         : std_logic_vector(7 downto 0);
13         -- Array with values -> 00: unsigned integer, 01: signed integer, 10: float
14         SUPL_TABLE_TYPE_G   : supl_table_type_t;
15         -- Array with values -> From 0 to 15 (check Table 3-1)
16         SUPL_TABLE_PURPOSE_G: supl_table_purpose_t;
17         -- Array with values -> 00: zero-dimensional, 01: one-dimensional, 10:
           two-dimensional-zx, 11: two-dimensional-yx
18         SUPL_TABLE_STRUCT_G : supl_table_struct_t;
19         -- Array with values -> Supplementary User-Defined Data
20         SUPL_TABLE_UDATA_G  : supl_table_udata_t
21     );
22     port (
23         md_img_width_o      : out integer;
24         md_img_data_o       : out std_logic_vector(1023 downto 0)
25     );
26 end entity metadata_img;
27
28 architecture behavioural of metadata_img is
29     -- Returns the size of a specific Table Data Subblock
30     function get_tdata_size(tdata_type_in : in std_logic_vector; tdata_struct_in :
31         in std_logic_vector; index_in : in integer) return integer is
32
33         -- Creates the Table Data Subblock
34         function create_tdata_subblock(tdata_type_in : in std_logic_vector;
35             tdata_struct_in : in std_logic_vector; index_in : in integer) return
36             std_logic_vector is
37
38         -- Returns the total size of all instantiated Supplementary Inforamtion Tables
39         function get_supl_tables_size(supl_tables_in : in mdata_img_supl_info_arr_t)
40             return integer is
41             variable supl_tables_size_v : integer := 0;
42         begin
43             for i in 0 to (supl_tables_in'length-1) loop
44                 supl_tables_size_v := supl_tables_size_v + supl_tables_in(i).total_width;
45             end loop;
46
47             return supl_tables_size_v;
48         end function get_supl_tables_size;
49
50         -- Creates the Supplementary Inforamtion Tables structure
51         function create_supl_tables(num_tables_in : in integer) return
52             mdata_img_supl_info_arr_t is
53             variable supl_tables_v : mdata_img_supl_info_arr_t(0 to num_tables_in-1);
54             variable tdata_size_v : integer := 0;
55         begin
56             for i in 0 to (num_tables_in-1) loop
57                 -- Compiler requires a fixed size for "table_data_subblock", so such
58                 size is managed independently
59                 tdata_size_v := get_tdata_size(SUPL_TABLE_TYPE_G(i),
60                     SUPL_TABLE_STRUCT_G(i), i);
61
62                 -- Supplementary table is filled
63                 supl_tables_v(i).table_type          := SUPL_TABLE_TYPE_G(i);
64                 supl_tables_v(i).reserved_1         := (others => '0');
65                 supl_tables_v(i).table_purpose        := SUPL_TABLE_PURPOSE_G(i);
66                 supl_tables_v(i).reserved_2         := (others => '0');
67                 supl_tables_v(i).table_structure    := SUPL_TABLE_STRUCT_G(i);
68             end loop;
69         end function create_supl_tables;
70
71     begin
72         md_img_width_o <= mdata_img_supl_info_arr_t(0).total_width;
73         md_img_data_o <= create_supl_tables(mdata_img_supl_info_arr_t(0).num_tables);
74     end architecture behavioural;
```

```

61         suppl_tables_v(i).reserved_3           := (others => '0');
62         suppl_tables_v(i).suppl_user_def_data  := SUPL_TABLE_UDATA_G(i);
63         suppl_tables_v(i).table_data_size     := tdata_size_v;
64         suppl_tables_v(i).table_data_subblock :=
65         create_tdata_subblock(SUPL_TABLE_TYPE_G(i), SUPL_TABLE_STRUCT_G(i), i);
66         suppl_tables_v(i).total_width         := 16 + tdata_size_v;
67     end loop;
68     return suppl_tables_v;
69 end function create_supl_tables;
70
71 -- Auxiliary constant to avoid the array below fail if "TAU_C=0"
72 constant TAU_0_C : integer := iif(TAU_C < 1, 1, TAU_C);
73 -- Record "Supplementary Information" structure from "Image Metadata" (Table 5-4)
74 -- NOTE: If TAU_C=0, array will be built anyway (so no compilation problems),
75 -- but will not be serialized in the corresponding function...
76 constant MDATA_IMG_SUPPL_INFO_ARR_C : mdata_img_suppl_info_arr_t(0 to TAU_0_C-1)
77 := create_supl_tables(TAU_0_C);
78
79 -- Record "Essential" sub-structure from "Image Metadata" (Table 5-3)
80 constant MDATA_IMG_ESSEN_C : mdata_img_essential_t := (
81     udef_data           => UDEF_DATA_G,
82     x_size              => std_logic_vector(to_unsigned(NX_C mod (2**16), 16)),
83     y_size              => std_logic_vector(to_unsigned(NY_C mod (2**16), 16)),
84     z_size              => std_logic_vector(to_unsigned(NZ_C mod (2**16), 16)),
85     smpl_type           => (others => SMPL_TYPE_G),
86     reserved_1          => (others => '0'),
87     larg_dyn_rng_flag   => (others => iif(D_C > 16, '1', '0')),
88     dyn_range           => std_logic_vector(to_unsigned(D_C mod (2**4), 4)),
89     smpl_enc_order      => (others => iif(SMPL_ORDER_G = BSQ_C, '1', '0')),
90     sub_frm_intlv_depth => std_logic_vector(to_unsigned(iif(SMPL_ORDER_G =
91     BSQ_C, 0, iif(SMPL_ORDER_G = BIL_C, M_BIL_C, M_BIP_C)) mod (2**16), 16)),
92     reserved_2          => (others => '0'),
93     out_word_size       => std_logic_vector(to_unsigned(B_C mod (2**3), 3)),
94     entropy_coder_type  => ENCODER_TYPE_G,
95     reserved_3          => (others => '0'),
96     quant_fidel_ctrl_mth=> FIDEL_CTRL_TYPE_G,
97     reserved_4          => (others => '0'),
98     suppl_info_table_cnt => std_logic_vector(to_unsigned(TAU_C, 4)),
99     total_width         => 96 -- The previous fields are a total of: 12 bytes
100     * 8 bits/byte = 96 bits
101 );
102
103 -- Record "Image Metadata" structure (Table 5-2)
104 -- NOTE: Do not confuse "encoded with n-bits" vs "encoded mod 2**n with n-bits"!!
105 constant MDATA_IMG_C : mdata_img_t := (
106     essential           => MDATA_IMG_ESSEN_C,
107     suppl_info_arr      => MDATA_IMG_SUPPL_INFO_ARR_C,
108     total_width         => MDATA_IMG_ESSEN_C.total_width + iif(TAU_C > 0,
109     get_supl_tables_size(MDATA_IMG_SUPPL_INFO_ARR_C), 0)
110 );
111
112 -- Image Metadata generation
113 constant OUT_MD_IMG_WIDTH_C : integer := MDATA_IMG_C.total_width;
114 constant OUT_MD_IMG_DATA_C : std_logic_vector(OUT_MD_IMG_WIDTH_C-1 downto 0) :=
115 serial_mdata_img(MDATA_IMG_C);
116
117 begin
118     -- Outputs
119     md_img_width_o      <= OUT_MD_IMG_WIDTH_C;
120     md_img_data_o(md_img_data_o'length-1 downto OUT_MD_IMG_WIDTH_C) <= (others =>
121     '0');
122     md_img_data_o(OUT_MD_IMG_WIDTH_C-1 downto 0) <= OUT_MD_IMG_DATA_C;
123 end architecture behavioural;

```

## 10.16 Parallel Synchronous FIFOs IP source code

```
1  library xpm;
2  use xpm.vcomponents.all;
3
4  entity parallel_sync_fifos is
5      generic (
6          SMPL_ORDER_G      : std_logic_vector(1 downto 0);
7          DATA_WIDTH_G     : integer;
8          DATA_DEPTH_G    : integer;
9          NUM_PAR_FIFOS_G  : integer
10     );
11     port (
12         clock_i           : in  std_logic;
13         reset_i           : in  std_logic;
14
15         enable_i          : in  std_logic;
16         enable_o          : out std_logic;
17         img_coord_i      : in  img_coord_t;
18         img_coord_o      : out img_coord_t;
19
20         read_fifo_i       : in  std_logic;
21         init_fifo_o       : out std_logic;
22
23         data_arr_i        : in  array_slv_t(0 to NUM_PAR_FIFOS_G-1)(DATA_WIDTH_G-1
24             downto 0);
25         data_arr_o        : out array_slv_t(0 to NUM_PAR_FIFOS_G-1)(DATA_WIDTH_G-1
26             downto 0)
27     );
28 end entity parallel_sync_fifos;
29
30 architecture behavioural of parallel_sync_fifos is
31     begin
32         g_sync_fifo_data : for i in 0 to NUM_PAR_FIFOS_G-1 generate
33             i_sync_fifo_data : xpm_fifo_sync
34                 generic map()
35                 port map(
36                     sleep           => '0',
37                     rst             => reset_i,
38                     wr_clk          => clock_i,
39                     wr_en           => wr_en_data_arr_s(i),
40                     din             => data_arr_i(i),
41                     full           => full_data_arr_s(i),
42                     prog_full      => open,
43                     wr_data_count  => wr_data_count_arr_s(i),
44                     overflow       => open,
45                     wr_rst_busy    => wr_rst_busy_data_arr_s(i),
46                     almost_full   => open,
47                     wr_ack         => open,
48                     rd_en         => rd_en_data_arr_s(i),
49                     dout           => data_arr_s(i),
50                     empty         => empty_data_arr_s(i),
51                     prog_empty     => open,
52                     rd_data_count  => rd_data_count_arr_s(i),
53                     underflow     => open,
54                     rd_rst_busy    => rd_rst_busy_data_arr_s(i),
55                     almost_empty  => open,
56                     data_valid    => open,
57                     injectdbiterr  => '0',
58                     injectsbiterr  => '0',
59                     sbiterr        => open,
60                     dbiterr        => open
61                 );
62         wr_en_data_arr_s(i) <= '1' when (wr_rst_busy_data_arr_s(i) = '0' and
63             full_data_arr_s(i) = '0') else '0';
64         rd_en_data_arr_s(i) <= '1' when (rd_rst_busy_data_arr_s(i) = '0' and
65             empty_data_arr_s(i) = '0' and read_fifo_i = '1') else '0';
66     end generate g_sync_fifo_data;
67
68     i_sync_fifo_coord : xpm_fifo_sync
69         generic map()
70         port map(
71             sleep           => '0',
72             rst             => reset_i,
73             wr_clk          => clock_i,
74             wr_en           => enable_i,
75             din             => img_coord_i,
76             full           => enable_o,
77             prog_full      => open,
78             wr_data_count  => img_coord_o,
79             overflow       => open,
80             wr_rst_busy    => open,
81             almost_full   => open,
82             wr_ack         => open,
83             rd_en         => read_fifo_i,
84             dout           => init_fifo_o,
85             empty         => open,
86             prog_empty     => open,
87             rd_data_count  => open,
88             underflow     => open,
89             rd_rst_busy    => open,
90             almost_empty  => open,
91             data_valid    => open,
92             injectdbiterr  => open,
93             injectsbiterr  => open,
94             sbiterr        => open,
95             dbiterr        => open
96         );
97 end architecture behavioural;
```

```

70     wr_en           => wr_en_coord_s,
71     din            => serial_img_coord_in_s,
72     full           => full_coord_s,
73     prog_full      => open,
74     wr_data_count  => wr_coord_count_s,
75     overflow       => open,
76     wr_rst_busy    => wr_rst_busy_coord_s,
77     almost_full   => open,
78     wr_ack         => open,
79     rd_en          => rd_en_coord_s,
80     dout           => serial_img_coord_out_s,
81     empty          => empty_coord_s,
82     prog_empty     => open,
83     rd_data_count  => rd_coord_count_s,
84     underflow      => open,
85     rd_rst_busy    => rd_rst_busy_coord_s,
86     almost_empty  => open,
87     data_valid     => open,
88     injectdbiterr => '0',
89     injectsbiterr => '0',
90     sbiterr        => open,
91     dbiterr        => open
92 );
93 wr_en_coord_s <= '1' when (wr_rst_busy_coord_s = '0' and full_coord_s = '0')
94   else '0';
95 rd_en_coord_s <= '1' when (rd_rst_busy_coord_s = '0' and empty_coord_s = '0' and
96   read_fifo_i = '1') else '0';

serial_img_coord_in_s(FIFO_X_COORD_WIDTH_C+FIFO_Y_COORD_WIDTH_C+FIFO_Z_COORD_WIDTH
_C-1 downto FIFO_Y_COORD_WIDTH_C+FIFO_Z_COORD_WIDTH_C) <=
std_logic_vector(to_signed(img_coord_i.x, FIFO_X_COORD_WIDTH_C));
97 serial_img_coord_in_s(FIFO_Y_COORD_WIDTH_C+FIFO_Z_COORD_WIDTH_C-1 downto
FIFO_Z_COORD_WIDTH_C) <= std_logic_vector(to_signed(img_coord_i.y,
FIFO_Y_COORD_WIDTH_C));
98 serial_img_coord_in_s(FIFO_Z_COORD_WIDTH_C-1 downto 0) <=
std_logic_vector(to_signed(img_coord_i.z, FIFO_Z_COORD_WIDTH_C));
99 extr_img_coord_s <= (
100   x =>
to_integer(signed(serial_img_coord_out_s(FIFO_X_COORD_WIDTH_C+FIFO_Y_COORD_WID
TH_C+FIFO_Z_COORD_WIDTH_C-1 downto
FIFO_Y_COORD_WIDTH_C+FIFO_Z_COORD_WIDTH_C))),
101   y =>
to_integer(signed(serial_img_coord_out_s(FIFO_Y_COORD_WIDTH_C+FIFO_Z_COORD_WID
TH_C-1 downto FIFO_Z_COORD_WIDTH_C))),
102   z => to_integer(signed(serial_img_coord_out_s(FIFO_Z_COORD_WIDTH_C-1 downto
0))),
103   t =>
to_integer(signed(serial_img_coord_out_s(FIFO_Y_COORD_WIDTH_C+FIFO_Z_COORD_WID
TH_C-1 downto FIFO_Z_COORD_WIDTH_C))) * NX_C +
to_integer(signed(serial_img_coord_out_s(FIFO_X_COORD_WIDTH_C+FIFO_Y_COORD_WID
TH_C+FIFO_Z_COORD_WIDTH_C-1 downto
FIFO_Y_COORD_WIDTH_C+FIFO_Z_COORD_WIDTH_C)))
104 );
105
106 enable_o <= enable_s;
107 img_coord_o <= extr_img_coord_s when read_fifo_i = '1' else
reset_img_coord(SMPL_ORDER_G);
108 data_arr_o <= data_arr_s when read_fifo_i = '1' else (others => (others
=> '0'));
109 init_fifo_o <= not(and(wr_rst_busy_data_arr_s) and and(rd_rst_busy_data_arr_s)
and wr_rst_busy_coord_s and rd_rst_busy_coord_s);
110 end architecture behavioural;

```

## 10.17 Sample-Adaptive GPO2 Code IP source code

```
1  entity smpl_adap_gpo2_code is
2  generic (
3      SMPL_ORDER_G      : std_logic_vector(1 downto 0);
4      MAX_NUM_BITS_G    : integer;
5      FIDEL_CTRL_TYPE_G : std_logic_vector(1 downto 0);
6      ABS_ERR_BAND_TYPE_G : std_logic;
7      REL_ERR_BAND_TYPE_G : std_logic
8  );
9  port (
10     clock_i      : in  std_logic;
11     reset_i      : in  std_logic;
12
13     enable_i     : in  std_logic;
14     enable_o     : out std_logic;
15     img_coord_i  : in  img_coord_t;
16     img_coord_o  : out img_coord_t;
17
18     err_lim_upd_en_i : in  std_logic;
19     err_lim_val_i   : in  std_logic_vector(D_C-1 downto 0);
20     data_mp_quan_i  : in  std_logic_vector(D_C-1 downto 0);
21
22     accu_i       : in  std_logic_vector(D_C*2-1 downto 0);
23     counter_i    : in  std_logic_vector(D_C-1 downto 0);
24     enc_num_bits_o : out std_logic_vector(MAX_NUM_BITS_G-1 downto 0);
25     codeword_o   : out std_logic_vector(Umax_C+D_C-1 downto 0)
26 );
27 end entity smpl_adap_gpo2_code;
28
29 architecture behavioural of smpl_adap_gpo2_code is
30 begin
31     p_calc_kz : process(clock_i) is
32         variable comp1_v : std_logic_vector(D_C*2-1 downto 0) := (others => '0');
33         variable comp2_v : std_logic_vector(D_C*2-1 downto 0) := (others => '0');
34         variable comp3_v : std_logic_vector(D_C*2-1 downto 0) := (others => '0');
35     begin
36         if rising_edge(clock_i) then
37             if (reset_i = '1') then
38                 comp1_v := (others => '0');
39                 comp2_v := (others => '0');
40                 comp3_v := (others => '0');
41                 log2_s  <= 0;
42                 kz_s   <= 0;
43             else
44                 if (enable_i = '1' and err_lim_upd_en_i = '0') then
45                     comp1_v := std_logic_vector(to_unsigned(division_down(49*to_integer(
46                         unsigned(counter_i)), 2**7), D_C*2));
47                     comp2_v := std_logic_vector(resize(unsigned(accu_i) + unsigned(comp1_v),
48                         D_C*2));
49                     comp3_v := std_logic_vector(resize(unsigned(comp2_v) / resize(unsigned(
50                         counter_i), D_C*2), D_C*2));
51                     log2_s  <= log2(to_integer(unsigned(comp3_v)) + 1);
52                     kz_s   <= iif(log2_s <= 0, 0, iif(log2_s > D_C-2, D_C-2, log2_s-1));
53                 end if;
54             end if;
55         end process p_calc_kz;
56
57     p_calc_codeword : process(clock_i) is
58         variable var_length_v : integer range 0 to 2**D_C := 0;
59         variable err_lim_type_v : std_logic := '0';
60         variable data_mp_quan_v : std_logic_vector(D_C-1 downto 0) := (others => '0');
61         variable codeword_v : std_logic_vector(Umax_C+D_C-1 downto 0) := (others => '0');
62     begin
63         if rising_edge(clock_i) then
64             if (reset_i = '1') then
65                 var_length_v := 0;
66                 err_lim_type_v := '0';
67                 codeword_v := (others => '0');
68                 err_lim_cnt_s <= 0;
69                 enc_num_bits_s <= (others => '0');
70                 codeword_s <= (others => '0');
71                 data_mp_quan_v := (others => '0');
72             else
73

```



```

71     if (img_coord2_s = end_img_coord) then
72         enc_num_bits_s <= (others => '0');
73         codeword_v      := (others => '0');
74     end if;
75
76     if (enable1_s = '1') then
77         codeword_v := (others => '0');
78         -- Mapped quantizer indexes ("z(t)") are encoded here!
79         if (err_lim_upd_en_s = '0') then
80             err_lim_cnt_s <= 0;
81             if (img_coord1_s.t = 0) then
82                 enc_num_bits_s <= std_logic_vector(to_unsigned(D_C,
83                     MAX_NUM_BITS_G));
84                 codeword_v      := data_mp_quan_s & (Umax_C-1 downto 0 => '0');
85             else -- img_coord1_s.t > 0
86                 var_length_v    := to_integer(shift_usg(unsigned(data_mp_quan_s),
87                     -kz_s));
88                 if (var_length_v < Umax_C) then
89                     enc_num_bits_s <= std_logic_vector(to_unsigned(var_length_v +
90                         1 + kz_s, MAX_NUM_BITS_G));
91                     data_mp_quan_v := data_mp_quan_s sll (D_C - kz_s);
92                     codeword_v    := (var_length_v-1 downto 0 => '0') & '1' &
93                         data_mp_quan_v & (Umax_C-var_length_v-2 downto 0 => '0');
94                 else -- var_length_v >= Umax_C
95                     enc_num_bits_s <= std_logic_vector(to_unsigned(Umax_C + D_C,
96                         MAX_NUM_BITS_G));
97                     codeword_v    := (Umax_C-1 downto 0 => '0') & data_mp_quan_s;
98                 end if;
99             end if;
100         end if;
101     else -- Error limit values are encoded here!
102         if (FIDEL_CTRL_TYPE_G = ABS_ERR_C) then
103             enc_num_bits_s <= std_logic_vector(to_unsigned(DA_C,
104                 MAX_NUM_BITS_G));
105             codeword_v(DA_C-1 downto 0) := err_lim_val_s(DA_C-1 downto 0);
106         elsif (FIDEL_CTRL_TYPE_G = REL_ERR_C) then
107             enc_num_bits_s <= std_logic_vector(to_unsigned(DR_C,
108                 MAX_NUM_BITS_G));
109             codeword_v(DR_C-1 downto 0) := err_lim_val_s(DR_C-1 downto 0);
110         elsif (FIDEL_CTRL_TYPE_G = ABS_REL_C) then
111             if ((ABS_ERR_BAND_TYPE_G = '1' and err_lim_cnt_s = 0) or (
112                 ABS_ERR_BAND_TYPE_G = '0' and err_lim_cnt_s < NZ_C)) then
113                 err_lim_type_v := '1';
114             else
115                 err_lim_type_v := '0';
116             end if;
117             err_lim_cnt_s <= err_lim_cnt_s + 1;
118             if (err_lim_type_v = '1') then
119                 enc_num_bits_s <= std_logic_vector(to_unsigned(DA_C,
120                     MAX_NUM_BITS_G));
121                 codeword_v(DA_C-1 downto 0) := err_lim_val_s(DA_C-1 downto 0);
122             else -- err_lim_type_v = '0'
123                 enc_num_bits_s <= std_logic_vector(to_unsigned(DR_C,
124                     MAX_NUM_BITS_G));
125                 codeword_v(DR_C-1 downto 0) := err_lim_val_s(DR_C-1 downto 0);
126             end if;
127         else
128             enc_num_bits_s <= (others => '0');
129             codeword_v      := (others => '0');
130         end if;
131     end if;
132     codeword_s <= codeword_v;
133 end process p_calc_codeword;
134
135 enable_o <= enable2_s;
136 img_coord_o <= img_coord2_s;
137 enc_num_bits_o <= enc_num_bits_s;
138 codeword_o <= codeword_s;
139 end architecture behavioural;

```

## 10.18 Adder IP Python script source code

```
1  # ***** VUNIT AND LIBRARY INITIALIZATION *****
2
3  # Load required libraries from VUnit
4  from os.path import join, dirname
5  # The public interface of VUnit
6  from vunit import VUnit
7  # Computes the cartesian product of input iterables
8  from itertools import product
9  # Load required functions for testcase files load
10 from os import listdir
11
12 # Returns the directory name where the present file (run.py) is located
13 root = dirname(__file__)
14
15 # Create VUnit instance by parsing command line arguments
16 ui = VUnit.from_argv()
17
18 # Add random numbers generation package
19 ui.add_random()
20 # Add verification component library
21 ui.add_verification_components()
22 # Add communication package
23 ui.add_com()
24
25 # Create library 'vunit_lib'
26 vunit_lib = ui.library("vunit_lib")
27
28 # Add all package files
29 vunit_lib.add_source_files(join(root, "../../../Image/_packages/*.vhd"))
30 vunit_lib.add_source_files(join(root, "../../../Predictor/_packages/*.vhd"))
31 vunit_lib.add_source_files(join(root, "../../../Encoder/_packages/*.vhd"))
32
33 # Add all sources files from Predictor IP
34 vunit_lib.add_source_files(join(root, "../../../Image/image_coord_control.vhd"))
35 vunit_lib.add_source_files(join(root, "../../../*.vhd"))
36
37 # Add testbench file from Predictor IP
38 vunit_lib.add_source_files(join(root, "tb_adder.vhd"))
39
40 # ***** FUNCTIONS *****
41
42 # To encode the parameters, the script must contain the encode function
43 def encode(tb_cfg):
44     return ",".join(["%s:%s" % (key, str(tb_cfg[key])) for key in tb_cfg])
45
46 # A list of parameters are defined, then saved and finally encoded to be in the
47 # testbench
48 # NOTE: Less than 2 parameters makes the whole system fail...
49 def gen_adder_tests(obj, smpl_type, smpl_order):
50     for smpl_type, smpl_order in product(smpl_type, smpl_order):
51         tb_cfg = dict(
52             SMPL_TYPE_PY=smpl_type,
53             SMPL_ORDER_PY=smpl_order
54         )
55         config_name = encode(tb_cfg)
56         obj.add_config(name=config_name, generics=dict(encoded_tb_cfg=encode(tb_cfg)))
57
58 # ***** GENERATE TESTBENCHES *****
59
60 # Everytime a new TB is here requested (can be the same with different parameters),
61 # all test cases in the VHDL testbench file will be executed again.
62 tb_adder = vunit_lib.test_bench("tb_adder")
63 for test in tb_adder.get_tests():
64     gen_adder_tests(test, ['0', '1'], [0])
65
66 # ***** MAIN FUNCTION *****
67 ui.main()
```

## 10.19 Adder IP VUnit testbench source code

```

1  entity tb_adder is
2      generic (
3          encoded_tb_cfg : string;
4          runner_cfg     : string
5      );
6  end entity tb_adder;
7
8  architecture behavioural of tb_adder is
9      -- Record type to pack all signals (from Python script) together
10     type tb_cfg_t is record
11         SMPL_TYPE_G : std_logic;
12         SMPL_ORDER_G : std_logic_vector(1 downto 0);
13     end record tb_cfg_t;
14
15     -- Function to decode the Python signals and connect them into the VHDL testbench
16     impure function decode(encoded_tb_cfg : string) return tb_cfg_t is
17     begin
18         return (
19             SMPL_TYPE_G => std_logic'value(get(encoded_tb_cfg, "SMPL_TYPE_PY")),
20             SMPL_ORDER_G =>
21                 std_logic_vector(to_unsigned(integer'value(get(encoded_tb_cfg,
22                 "SMPL_ORDER_PY")), 2))
23         );
24     end function decode;
25
26 begin
27     reset_s <= '0' after 20 ns;
28     clock_s <= not clock_s after 6.4 ns;    -- Main clock frequency: 78125000 Hz
29
30     -- Simulation MAIN process (if this one finishes, the whole simulation too)
31     test_runner : process is
32     begin
33         test_runner_setup(runner, runner_cfg);
34
35         while test_suite loop
36             if run("Top Adder Block") then
37                 info("Running test case = " & to_string(running_test_case));
38                 flag_start <= '1';
39             end if;
40
41             wait until (flag_stop = '1');
42         end loop;
43         test_runner_cleanup(runner);
44     end process;
45     test_runner_watchdog(runner, 1 ms);
46
47     -- Input signals stimulus calculation
48     p_inputs_update : process(clock_s) is
49     begin
50         if (rising_edge(clock_s)) then
51             if (reset_s = '1') then
52                 pred_en_s      <= '0';
53                 pred_coord_s  <= reset_img_coord(tb_cfg.SMPL_ORDER_G);
54                 data_s0_s     <= (others => '0');
55                 data_s3_1s    <= (others => '0');
56                 data_s4_lsb_1s <= '0';
57             else
58                 if (flag_start = '1') then
59                     if ((img_coord_out_s.x < NX_C-1) or (img_coord_out_s.y < NY_C-1)
60                     or (img_coord_out_s.z < NZ_C-1)) then
61                         pred_en_s      <= enable_out_s;
62                         pred_coord_s  <= img_coord_out_s;
63
64                         if (data_s3_1s = (data_s3_1s'length-1 downto 0 => '1')) then
65                             data_s3_1s <= (others => '0');
66                         else
67                             data_s3_1s <=
68                                 std_logic_vector(resize(signed(data_s3_1s) +
69                                 to_signed(159, data_s3_1s'length), data_s3_1s'length));

```

```

65         end if;
66
67         if (data_s0_s = (data_s0_s'length-1 downto 0 => '1')) then
68             data_s0_s <= (others => '0');
69         else -- "data_s0_s" should update faster than
70             "data_s3_1s"
71             data_s0_s <= std_logic_vector(resize(signed(data_s0_s)
72             + to_signed(5, data_s0_s'length), data_s0_s'length));
73         end if;
74
75         if (img_coord_out_s.z = 0) then
76             data_s4_lsb_1s <= not data_s4_lsb_1s;
77         end if;
78     end if;
79 end if;
80 end process p_inputs_update;
81
82 p_stop_sim : process is
83 begin
84     wait until ((img_coord_out_s.x >= NX_C-1) and (img_coord_out_s.y >= NY_C-1)
85     and (img_coord_out_s.z >= NZ_C-1));
86     wait for 10 ns;
87     flag_stop <= '1';
88 end process p_stop_sim;
89
90 i_image_coord_control : image_coord_control
91 generic map(
92     SMPL_ORDER_G => tb_cfg.SMPL_ORDER_G
93 )
94 port map(
95     clock_i      => clock_s,
96     reset_i      => reset_s,
97
98     enable_i     => flag_start,
99     enable_o     => img_en_s,
100
101     image_end_i  => '0',
102     img_coord_o  => img_coord_in_s
103 );
104
105 i_adder : adder
106 generic map(
107     SMPL_ORDER_G => tb_cfg.SMPL_ORDER_G,
108     SMPL_TYPE_G  => tb_cfg.SMPL_TYPE_G
109 )
110 port map(
111     clock_i      => clock_s,
112     reset_i      => reset_s,
113
114     img_en_i     => img_en_s,
115     enable_o     => enable_out_s,
116     img_coord_i  => img_coord_in_s,
117     img_coord_o  => img_coord_out_s,
118
119     pred_en_i    => pred_en_s,
120     pred_coord_i => pred_coord_s,
121     data_s3_i    => data_s3_1s,
122     data_s3_o    => data_s3_2s,
123     data_s4_lsb_i => data_s4_lsb_1s,
124     data_s4_lsb_o => data_s4_lsb_2s,
125
126     data_s0_i    => data_s0_s,
127     en_res_o     => en_res_s,
128     data_res_o   => data_res_s
129 );
130 end architecture behavioural;

```

## 10.20 Simulations Bash source code

```
1  echo -e -n "  
2  Select a block to validate:  
3  - Top entity:          0  
4  - Image block:        1  
5  - Predictor block:    2  
6    - Adder sub-block:  3  
7    - Quantizer sub-block: 4  
8    - Mapper sub-block:  5  
9    - Sample Repr. sub-block: 6  
10   - Prediction sub-block: 7  
11  - Encoder block:      8  
12    - Enc. Header sub-block: 9  
13    - Enc. Body sub-block: 10  
14      - Sample-Adapt. Coder: 11  
15      - Hybrid Coder:      12  
16    - Packer sub-block: 13  
17  - All blocks:        14  
18  Clean compiled files: 15  
19  \n"  
20  
21  unset option  
22  until [[ $option == +([0-9]) && $option -le 15 ]]; do  
23    read -r -p "Enter a valid number: " option  
24  done  
25  
26  unset path; unset place  
27  if [[ $option -eq 0 ]]; then  
28    path="./TOP/"; place="TOP"  
29  elif [[ $option -eq 1 ]]; then  
30    path="./Image/"; place="IMAGE"  
31  elif [[ $option -eq 2 ]]; then  
32    path="./Predictor/"; place="PREDICTOR"  
33  elif [[ $option -eq 3 ]]; then  
34    path="./Predictor/adder/"; place="PRED: ADDER"  
35  elif [[ $option -eq 4 ]]; then  
36    path="./Predictor/quantizer/"; place="PRED: QUANTIZER"  
37  elif [[ $option -eq 5 ]]; then  
38    path="./Predictor/mapper/"; place="PRED: MAPPER"  
39  elif [[ $option -eq 6 ]]; then  
40    path="./Predictor/sample_representative"; place="PRED: SAMPLE REPR."  
41  elif [[ $option -eq 7 ]]; then  
42    path="./Predictor/prediction/"; place="PRED: PREDICTION"  
43  elif [[ $option -eq 8 ]]; then  
44    path="./Encoder/"; place="ENCODER"  
45  elif [[ $option -eq 9 ]]; then  
46    path="./Encoder/header/"; place="ENC: ENCODER HEADER"  
47  elif [[ $option -eq 10 ]]; then  
48    path="./Encoder/body/"; place="ENC: ENCODER BODY"  
49  elif [[ $option -eq 11 ]]; then  
50    path="./Encoder/body/sample_adaptive_coder"; place="ENC. BODY: SMPL-ADAPT. CODER"  
51  elif [[ $option -eq 12 ]]; then  
52    path="./Encoder/body/hybrid_coder"; place="ENC. BODY: HYBRID CODER"  
53  elif [[ $option -eq 13 ]]; then  
54    path="./Encoder/packer/"; place="ENC: PACKER"  
55  elif [[ $option -eq 14 ]]; then  
56    path="."; place="ALL"  
57  else  
58    rm -rf vunit_out/;  
59    echo -e "\n --> FOLDER 'vunit_out' DELETED. EXITING... ";  
60    exit 0;  
61  fi; echo -e " *** VALIDATING -> $place IP(s) "  
62  
63  for file in $(find $path -name 'run_*.py'); do  
64    echo -e "\n --> EXECUTING TESTBENCH \"${file#*simulation/}\" \n"  
65    python $file ||  
66    { echo -e "\n PROBLEM FOUND ON TESTBENCH \"${file#*simulation/}\".  
        VALIDATION PROCESS ABORTED..."; exit 1; }  
67  done
```

## 10.21 Xilinx Vivado TCL framework source code

```
1  ## PROJECT VARIABLES
2
3  # Set the project name
4  set xil_proj_name "ccsds123issue2_project"
5
6  # Set the default library
7  set default_library "work"
8
9  # Set the SoC HW reference for the project
10 set fpga_board_ref xc7z020clg400-1
11 # other references: xczu9eg-ffvc900-1-e
12 set ev_board_ref zcu104
13
14 # Project main HDL to use. Possible values: "VHDL", "Verilog" or "Mixed"
15 set project_language "VHDL"
16 # Specific language for both the source/simulation and constraint files
17 set sources_language "VHDL 2008"
18 set constr_language "XDC"
19
20 # The number of CPU cores to use for the "Synthesis" and "Implementation" processes
21 set num_cores 6
22 # Enable/Disable the Synthesis Out-Of-Context (disabled makes process faster and
23 # more efficient)
24 set synth_ooc false
25 # To automatically perform the "Synthesis" process
26 set synth_enable true
27 # To automatically perform the "Implementation" process (it also includes the
28 # bitstream generation)
29 set imple_enable true
30
31 ## PROJECT CREATION AND CONFIGURATION
32 puts " -----> INFO: Project is created and configured!"
33
34 # Set the reference directory for source file relative paths (by default the value
35 # is script directory path)
36 set origin_dir "."
37
38 # Set the directory path for the original project from where this script was exported
39 set orig_proj_dir "[file normalize "${origin_dir}/${xil_proj_name}]"
40
41 # Create project
42 create_project -force ${xil_proj_name} ./${xil_proj_name} -part ${fpga_board_ref}
43
44 # Set the directory path for the new project
45 set proj_dir [get_property directory [current_project]]
46
47 # Set project properties
48 set obj [current_project]
49
50 ## SOURCING ALL TCL FILES
51 puts " -----> INFO: All TCL files are sourced!"
52 source scripts/source_files.tcl
53 source scripts/constraint_files.tcl
54 source scripts/artifacts_generation.tcl
55
56 ## SOURCE FILESET CREATION
57 puts " -----> INFO: Source files are introduced into the project!"
58
59 # Create 'sources_1' fileset (if not found)
60 if {[string equal [get_filesets -quiet sources_1] ""]} {
61     create_fileset -srcset sources_1
62 }
63
64 # Set 'sources_1' fileset object (first element is set later as the top entity)
65 add_files -norecurse -fileset [get_filesets sources_1] ${source_files_list}
66
67 # Name of the top entity file (the first element from "source_files_list") is
68 # extracted
69 set top_entity_name [file rootname [file tail [lindex ${source_files_list} 0]]]
70
71 foreach {source_file} ${source_files_list} {
72     set file_obj [get_files -of_objects [get_filesets sources_1] [list
73         "$source_file"]]
```

```

69     set_property -name "file_type" -value ${sources_language} -objects ${file_obj}
70     set_property -name "library" -value ${default_library} -objects ${file_obj}
71     set_property -name "used_in_synthesis" -value true -objects ${file_obj}
72     set_property -name "used_in_simulation" -value true -objects ${file_obj}
73 }
74
75 # Set 'sources_1' fileset properties (defining the first element from previous list
as the top entity)
76 set_property -name "top" -value ${top_entity_name} -objects [get_filesets sources_1]
77
78 ## CONSTRAINTS FILESET CREATION
79 puts " -----> INFO: Constraint files are introduced into the project!"
80
81 # Create 'constrs_1' fileset (if not found)
82 if {[string equal [get_filesets -quiet constrs_1] ""]} {
83     create_filesset -constrset constrs_1
84 }
85
86 # Set 'constrs_1' fileset object
87 # NOTE: List "constr_files_list" has previously been sourced from file
"constr_files.tcl"
88 add_files -norecurse -filesset [get_filesets constrs_1] ${constr_files_list}
89
90 foreach {constr_file} ${constr_files_list} {
91     set file_obj [get_files -of_objects [get_filesets constrs_1] [list
92     "$constr_file"]]
93     set_property -name "file_type" -value ${constr_language} -objects ${file_obj}
94     set_property -name "library" -value ${default_library} -objects ${file_obj}
95     set_property -name "used_in_synthesis" -value true -objects ${file_obj}
96     set_property -name "used_in_implementation" -value true -objects ${file_obj}
97 }
98
99 # Set 'constrs_1' fileset properties
set_property -name "target_part" -value "${fpga_board_ref}" -objects [get_filesets
constrs_1]
100
101 ## SIMULATION FILESET CREATION
102 puts " -----> INFO: Simulation files are introduced into the project!"
103
104 # Create 'sim_1' fileset (if not found)
105 if {[string equal [get_filesets -quiet sim_1] ""]} {
106     create_filesset -simset sim_1
107 }
108
109 # Set 'sim_1' fileset properties
110 set obj [get_filesets sim_1]
111 set_property -name "top" -value ${top_entity_name} -objects ${obj}
112 set_property -name "top_lib" -value ${default_library} -objects ${obj}
113
114 ## SYNTHESIS/IMPLEMENTATION + BITSTREAM/HW DESCRIPTION FILE GENERATION
115 puts " -----> INFO: Synthesis/Implementation processes are executed!"
116
117 # Xilinx Vivado "Synthesis" process execution
118 if { ${synth_enable} == true } {
119     run_synthesis ${synth_ooc} ${num_cores}
120 }
121
122 # Xilinx Vivado "Implementation" process execution (and .bit/.hdf files
generation)
123 if { ${imple_enable} == true } {
124     run_implementation ${num_cores} ${origin_dir} ${xil_proj_name}
125     ${top_entity_name}
126 }

```

## 10.22 Vivado Project Bash source code

```
1  #/bin/sh
2
3  # *****
4  # ***** CREATE AND/OR OPEN THE VIVADO PROJECT *****
5  # *****
6
7  # *****
8  # INSTRUCTIONS FOR EXECUTION:
9  # 1. Install "Xilinx Vivado 2019.1" (including it to the PATH as well).
10 # 2. Open a terminal and execute the present file --> "sh run_vivado_project.sh"
11 # 3. Follow instructions
12 # *****
13
14 # Displays a bunch of options, reads back the argument, and repeats if wrong value
15 echo -e -n "
16 Select an option:
17 - Generate Vivado project (batch mode): 1
18 - Generate Vivado project (GUI mode): 2
19 - Open generated Vivado project: 3
20 - Delete generated Vivado project: 4
21 - Run Vivado TCL shell: 5
22 \n"
23
24 # Infinite loop until a number from 1 to 5 is entered
25 unset option
26 until [[ $option == [1-5] ]]; do
27     read -r -p "Enter a valid number: " option
28 done
29
30 # Goes where the Vivado TCL framework is located
31 cd _vivado_framework/
32
33 # Executes the selected command
34 if [[ $option -eq 1 ]]; then
35     vivado -mode batch -source generate_project.tcl
36 elif [[ $option -eq 2 ]]; then
37     vivado -mode gui -source generate_project.tcl
38 elif [[ $option -eq 3 ]]; then
39     vivado -mode gui ccsdsl23issue2_project/ccsdsl23issue2_project.xpr
40 elif [[ $option -eq 4 ]]; then
41     rm -rf ccsdsl23issue2_project/; rm -rf artifacts/
42     rm -rf .Xil/; rm *.log; rm *.jou; rm *.str; rm *.zip
43 else
44     vivado -mode tcl
45 fi
46
47 # Comes back to the original directory
48 cd ..
```



# Bibliography

- [1] Lars Asplund. Vunit check library. [https://vunit.github.io/check/user\\_guide.html](https://vunit.github.io/check/user_guide.html). Accessed 02.10.2021.
- [2] Lars Asplund. Vunit communication library. [https://vunit.github.io/com/user\\_guide.html](https://vunit.github.io/com/user_guide.html). Accessed 02.10.2021.
- [3] Lars Asplund. Vunit run library. [https://vunit.github.io/run/user\\_guide.html](https://vunit.github.io/run/user_guide.html). Accessed 02.10.2021.
- [4] Lars Asplund. Vunit user guide. [https://vunit.github.io/user\\_guide.html#introduction](https://vunit.github.io/user_guide.html#introduction). Accessed 01.10.2021.
- [5] Lars Asplund. Vunit verification component library. [https://vunit.github.io/verification\\_components/user\\_guide.html](https://vunit.github.io/verification_components/user_guide.html). Accessed 02.10.2021.
- [6] Lars Asplund. What is vunit? <https://vunit.github.io/about.html>. Accessed 6.12.2021.
- [7] Roger Birkeland. Ntnu small satellite lab. <https://www.ntnu.edu/ie/smallsat/mission-hyper-spectral-camera>. Accessed 11.09.2020.
- [8] Roger Birkeland. Ntnu small satellite lab. <https://www.ntnu.edu/ie/smallsat/project-overview>. Accessed 11.09.2020.
- [9] Christoffer Boothby. *An implementation of a compression algorithm for hyperspectral images. A novelty of the CCSDS 123.0-B-2 standard*. NTNU, 2020.
- [10] CCSDS. *Low-complexity lossless and near-lossless multispectral and hyperspectral image compression*, 2 2012. v1.0.
- [11] CCSDS. *Lossless multispectral and hyperspectral image compression*, 5 2019. v1.0.
- [12] Andreas Deuter. *Slicing the V-Model – Reduced Effort, Higher Flexibility*. IEEE, 2013.
- [13] European Space Agency (ESA). The use of reprogrammable fpgas in space. [https://www.esa.int/Enabling\\_Support/Space\\_Engineering\\_Technology/Microelectronics/The\\_use\\_of\\_reprogrammable\\_FPGAs\\_in\\_space](https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Microelectronics/The_use_of_reprogrammable_FPGAs_in_space). Accessed 23.06.2021.

- [14] Joan Serra-Sagrista Ian Blanes. Aaron Kiely. Lucana Santos. Miguel Hernández. *The hybrid entropy encoder of CCSDS-123.0-B-2: Insights and decoding process*. IEEE, 2019.
- [15] Zeljko Zilic Ian Brynjolfson. Fpga clock management for low power. *Research Gate*, page 12, nov 1999.
- [16] Ricardo Jasinski. *Effective Coding with VHDL: Principles and best practice*. The MIT Press, 2016.
- [17] Johan Fjeldtvedt. Milica Orlandic. Tor Arne Johansen. *An Efficient Real-Time FPGA Implementation of the CCSDS-123 Compression Standard for Hyperspectral Images*. IEEE, 2018.
- [18] Johan Fjeldtvedt. Milica Orlandic. Tor Arne Johansen. *A Parallel FPGA Implementation of the CCSDS-123 Compression Algorithm*. Remote Sensing, 2019.
- [19] Carolina Santos José Manuel Amigo. Hyperspectral imaging. <https://www.sciencedirect.com/topics/computer-science/hyperspectral-image>. Accessed 04.09.2021.
- [20] S.Hauck K.Compton. Reconfigurable computing: A survey of systems and software. *ACM Computing Systems*, page 40, jun 2002.
- [21] Panagiotis Chatziantoniou. Antonis Tsigkanos. Nektarios Kranitis. *A high-performance RTL implementation of the CCSDS-123.0-B-2 hybrid entropy coder on a space-grade SRAM FPGA*. IEEE, 2020.
- [22] Mentor Graphics. *ModelSim Tutorial*, 1 2016. v10.5c.
- [23] Uwe Meyer-Baese. *Embedded Microprocessor System Design using FPGAs*. Springer, 2020.
- [24] Cristian Gil Morales. *CCSDS123 Issue 2 (FPGA) Implementation*. NTNU, 2020.
- [25] Daniel Bascones. Carlos Gonzalez. Daniel Mozos. *A real-time FPGA implementation of the full CCSDS 123.0-B-2 standard*. IEEE, 2021.
- [26] Panagiotis Chatziantoniou. Antonis Tsigkanos. Antonis Tsigkanos. Dimitris Theodoropoulos. Nektarios Kranitis. Antonios Paschalis. *An Efficient Architecture and High-Throughput Implementation of CCSDS-123.0-B-2 Hybrid Entropy Coder Targeting Space-Grade SRAM FPGA Technology*. IEEE, 2021.
- [27] Jim Lewis Peter Ashenden. *VHDL-2008: Just the New Stuff*. Elsevier, 2007.
- [28] Yubal Barrios. Antonio Sánchez. Raúl Guerra. Roberto Sarmiento. *Hardware Implementation of the CCSDS 123.0-B-2 Near-Lossless Compression Standard Following an HLS Design Methodology*. Remote Sensing, 2021.

- [29] Miguel Hernandez. Aaron Kiely. Matthew Klimesh. Ian Blanes. Jonathan Ligo. Enrico Magli. Joan Serra. *The CCSDS 123.0-B-2 “Low-complexity Lossless and Near-Lossless Multispectral and Hyperspectral Image Compression” standard, explained*. IEEE, 2021.
- [30] Xilinx. *Integrated Logic Analyzer*, 4 2016. v6.1.
- [31] Xilinx. *Virtual Input/Output*, 4 2018. v3.0.
- [32] Xilinx. *UltraScale Architecture Libraries Guide*, 5 2019. v2019.1.
- [33] Xilinx. *ZCU102 Evaluation Board*, 6 2019. v1.6.
- [34] Xilinx. *Vivado Design Suite User Guide: Implementation*, 8 2020. v2020.1.
- [35] Xilinx. *Vivado Design Suite User Guide: Synthesis*, 1 2020. v2019.2.
- [36] Xilinx. *Zynq UltraScale+ Device: Packaging and Pinouts*, 6 2020. v1.9.
- [37] Xilinx. *Zynq UltraScale+ MPSoC Data Sheet: Overview*, 5 2021. v1.9.

# 11 Declaration of Authorship

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references. I am aware that the thesis in digital form can be examined for the use of unauthorized aid and in order to determine whether the thesis as a whole or parts incorporated in it may be deemed as plagiarism. For the comparison of my work with existing sources I agree that it shall be entered in a database where it shall also remain after examination, to enable comparison with future theses submitted. Further rights of reproduction and usage, however, are not granted here. This thesis was not previously presented to another examination board and has not been published.

Cristian Gil Morales, ID: 500306

---

Student name and ID



Barcelona, 18 January 2022

---

City, date and signature



