Critical Link, LLC.

Bayer Demosaic IP Core Design

For Use with Various Camera Designs

Table of Contents

1 Introduction 5

2 Design Details 5

2.1 Scalable Architecture 5

2.1.1 Datapath Scalability 5

2.1.2 Interpolation Kernel Scalability 6

2.2 HDL Module Hierarchy 7

2.2.1 [Bayer_Demosaic_Registers] 7

2.2.2 [Bayer_Demosaic_Reset] 7

2.2.3 [Bayer_Demosaic_Parser] 8

2.2.4 [Bayer_Demosaic_Pipeline] 8

2.2.4.1 [Bayer_Demosaic_Window] 8

2.2.4.2 [Bayer_Kernel(Nearest_Neighbor)] 8

2.2.4.3 [Bayer_Kernel(Arch_Selector)] 8

2.2.4.4 [Bayer_Demosaic_Balance] 8

2.2.5 [Bayer_Demosaic_Encapsulator] 9

2.2.5.1 [Bayer_Demosaic_Elastic] 9

3 Register File 10

3.1 Control Register 11

3.2 IRQ Flags Register 12

3.3 IRQ Mask Register 13

3.4 Resolution Capture Register 14

3.5 Region-of-Interest (ROI) Capture Register 15

3.6 Index Capture Register 16

3.7 Timestamp Capture Register 17

3.8 White Balance [Red] Register 18

3.9 White Balance [Green] Register 19

3.10 White Balance [Blue] Register 20

4 Verification Environment 21

4.1 Test Factory 21

4.2 Golden Model 21

4.3 Docker Image 22

Index of Tables

Table 3.1: Bayer Demosaic Core Global Register Map 8

Table 3.1.1: Control Register Field Definitions 9

Table 3.2.1: IRQ Flag Register Field Definitions 10

Table 3.3.1: IRQ Mask Register Field Definitions 11

Table 3.4.1: Resolution Capture Register Field Definitions 12

Table 3.5.1: ROI Capture Register Field Definitions 13

Table 3.6.1: Index Capture Register Field Definitions 14

Table 3.7.1: Timestamp Capture Register Field Definitions 15

Table 3.8.1: White Balance [Red] Register Field Definitions 16

Table 3.8.2: Numerical Representation of White Balance Gain Values 16

Table 3.9.1: White Balance [Green] Register Field Definitions 17

Table 3.10.1: White Balance [Blue] Register Field Definitions 18

References

1. *Frame Data Packing Design : For Various Camera Systems. Critical Link, LLC – Draft Revision*

2. *Welcome to Cocotb's documentation! : https://media.readthedocs.org/pdf/cocotb/latest/cocotb.pdf*

1 Introduction

This document details the design and implementation of the Critical Link Bayer Demosaic core, and is principally intended to be of use to embedded systems engineers incorporating the core into a system-on-chip design.

The core implements a set of interpolation algorithms for estimating a source image from raw Bayer-patterned data, typically originating directly from one or more digital image sensor.

2 Design Details

The design of the core is broken down into subsections, each of which provides a brief treatment of its subject matter.

The text herein is not intended to be a thorough description of the core design, but rather a high-level overview; for those requiring a greater degree of detail, the VHDL and Python code involved is written to be 'literate' in its expression, in terms of the amount and detail of comments and adherence to documentation headers.

2.1 Scalable Architecture

The core implements its functionality in a few dimensions of scalability; these have been derived from its initial requirements specification, as well as some lightweight proactive future-proofing strategies.

2.1.1 Datapath Scalability

The core is capable of handling a range of two (2) to sixteen (16) pixels per input clock beat, per reference [1]. The upper end is not inherently limited by anything other than an arbitrary, integer-power-of-two constant in a package file. As such, it may be easily extended in the future. The verification environment is similarly capable of scaling in this dimension to address the need to validate any future expansions.

2.1.2 Interpolation Kernel Scalability

The initial specification for the core calls for two distinct, region-based interpolation modes:

* Nearest-neighbor (NN) interpolation of pels on the first two columns / rows

* Mixed bi-linear / linear interpolation for all other ‘core’ pels

This necessitates a certain amount of overhead infrastructure: logic for feeding coordinate and input pixels to multiple processing submodules, then multiplexing between their respective output while compensating for differences in systolic delay, properly selecting based upon image coordinates, etc.

Since this infrastructure must be built in any case, the design is implemented in an expressly-extensible manner, easing the definition and integration of additional interpolation kernels. An example future enhancement is the inclusion of a bi-cubic interpolation kernel, producing a higher-quality interpolation result for imagery containing a significant amount of high-spatial-frequency content. This addition relegates the bi-linear output to only those pels within the third and fourth columns / rows.

2.2 HDL Module Hierarchy

This subsection provides a hierarchical exploration of each of the submodules within the design, with at least a comment about the function and / or organization of each.

The top level of the hierarchy is embodied in the `bayer_demosaic` VHDL entity. This is designed for direct integration within Intel Platform Designer (formerly Altera Qsys), and is paired with a hardware TCL script for UI and build flow integration.

2.2.1 [Bayer_Demosaic_Registers]

Implementing the register file, this module is the internal sink for the Avalon-MM slave interface. A modest set of hardware registers are exposed by this logic, each of which are detailed in heading 3 .

Many of the hosted hardware registers are static, in that they are intended to be set prior to enabling the core (also accomplished via a register bit) and remain at their configured values indefinitely throughout normal operation. The exception to this is the mechanism for capturing header information from incoming frames, which involves some dynamic activity via control and interrupt flag bits.

2.2.2 [Bayer_Demosaic_Reset]

The reset module is a small structural component providing synchronized reset signals for each clock domain. Each reset is asserted asynchronously, and with level-sensitive behavior, in response to either assertion of its own domain's reset signal or by the deassertion of the "core enable" bit in the register file. Deassertion of the module's reset outputs, on the other hand, occurs synchronously to each output's respective clock domain (input or output). Reset outputs are also guaranteed to be asserted for four (4) cycles of their respective domain's clock.

Despite being generated by synchronous logic, these resets are still quite suitable (and intended) for use as asynchronous resets to logic in each respective clock domain. In general, the reset philosophy applied throughout the design of the core is that of global-synchronization, but locally-asynchronous reset.

Furthermore, each registered signal within the design has been critically analyzed to determine whether a reset is necessary at all – many signals (in particular very wide signals) within a signal processing datapath are inherently periodically "reset" as an outcome of their state space transitions. Consequently, routing resources within the target chip may be alleviated by not inferring resets for said signals.

2.2.3 [Bayer_Demosaic_Parser]

The parser is responsible for performing packet-level parsing of the incoming Avalon-ST Video stream. Headers and pixel packing are parsed per the specification embodied within reference [1], and extracted field values are strobed out for consumption by the register file as well as downstream pipeline logic. Extracted payload data is output with a qualifying valid strobe, effectively stripping it of the header beat(s).

2.2.4 [Bayer_Demosaic_Pipeline]

This is a structural component, aggregating the subcomponents which comprise the buffering, signal processing, and alignment / orientation logic for the interpolation pipeline. Raw, monochromatic input payload data from the parser enters the pipeline, and RGB output payload data is produced for consumption by the downstream encapsulator. A logic block multiplexes between interpolation kernels.

2.2.4.1 [Bayer_Demosaic_Window]

The window generator submodule is responsible for generating sliding “windows” of input pel data to downstream interpolation arithmetic. A few small submodules (not explicitly broken out herein) provide logic for tracking writes to input line buffers, generation of coordinate excitation to “slide” over previously-buffered input data, and read addressing / data multiplexing logic to circulate through the fixed number of line buffers. The output is an NxN matrix of input pel values for interpolation.

2.2.4.2 [Bayer_Kernel(Nearest_Neighbor)]

The nearest-neighbor interpolation kernel is implemented as a specific architecture of the generic `Bayer_Kernel` entity declaration. As described in the overview of the interpolation kernels in heading 2.1.2 .

2.2.4.3 [Bayer_Kernel(Arch_Selector)]

This implementation of the `Bayer_Kernel` entity performs a compile-time configurable selection of the interpolation kernel(s) to instantiate for processing of ‘core’ (non-boundary / NN-interpolated) pixel locations. The present incarnation of the core selects the linear / bi-linear interpolation kernel, `Bayer_Kernel(Bilinear)`.

2.2.4.4 [Bayer_Demosaic_Balance]

The last stage of the Bayer Demosaic core pipeline, this submodule is responsible for implementing the white-balancing operation. It accomplishes its function by applying a register-configurable fixed-point gain value in the range `[0.0, 1.0]` to all interpolated output pels, using a distinct gain value for each color plane.

2.2.5 [Bayer_Demosaic_Encapsulator]

Performing the inverse of the parser, the encapsulator is responsible for wrapping output payload data produced by the pipeline with headers, producing the ultimate output of the core. Header fields are replicated from the input parser, with the exception of ‘spare’ quadlets, which are output as zeros.

2.2.5.1 [Bayer_Demosaic_Elastic]

The encapsulator stage is also responsible for implementing a single-cycle `READY` latency for the Avalon-ST protocol. This requires that it be capable of absorbing short bursts of “inertia” from the processing pipeline, while also respecting the ready cycles of the downstream sink. This submodule implements this behavior, making use of a shallow FIFO to accommodate the variable latency.

3 Register File

This subsection documents the details of the register file implemented for control and supervision of the Bayer Demosaic core. Table provides a top-level register map, specifying base-address-relative byte offsets of each register, while the sub-sections contained within address the field details and / or semantics of each.

Byte Offset	Name	Description
0x00	Control Register	Global core control / status
...	...	
0x10	IRQ Flags Register	Vector of flag bits for interrupt sources
0x14	IRQ Mask Register	Vector of mask bits for interrupt sources
...	...	
0x20	Resolution Capture Register	Resolution of last captured frame header
0x24	ROI Capture Register	ROI of last captured frame header
0x28	Index Capture Register	Index of last captured frame header
0x2C	Timestamp Capture Register	Timestamp of last captured frame header
0x30	White Balance [Red] Register	White balance gain – red image plane
0x34	White Balance [Green] Register	White balance gain – green image plane
0x38	White Balance [Blue] Register	White balance gain – blue image plane

Table 1: Bayer Demosaic Core Global Register Map

In the table Table, as with ensuing subsections, any address or bit region designated with an ellipsis (...) is interpreted as a *reserved* region of register file memory resources. Unless otherwise specified, bits within reserved regions are read-only, returning logic zero ('0').

Hardware reset values for all non-reserved register bit / field locations are indicated below their table columns in red text. Unless otherwise specified, the reset values are only applied upon *hardware reset* of the Avalon-MM Slave interface (host) clock domain. Registers which support write-only semantics express their reset value as the 'don't-care' condition, indicated by a hyphen ('-'), while registers which are not initialized to default values by hardware reset are indicated as undefined ('U')

Access indicators for bits / fields are as follows:

```
R    Read-only

W    Write-only

R/W  Read / Write

R/C  Read / Clear
```

Any bit marked as R/C is cleared by writing a '1' to its bit position.

3.1 Control Register

CTRL_REG

Byte offset 0x00

Bit(s)	[31]	[30..3]	[2]	[1]	[0]
Access	R/W	...	R/W	R/W	W
Name	CORE_EN	...	COL_MODE	ROW_MODE	CAP_CTRL
	0		0	0	-

Bit(s)	Access	Name	Description
31	R/W	CORE_EN	Core Enable 0 = Disabled 1 = Enabled
2	R/W	COL_MODE	Input Bayer Pattern Column Mode 0 = First pel columns green 1 = First pel columns red or blue
1	R/W	ROW_MODE	Input Bayer Pattern Row Mode 0 = First pel row is green & red 1 = First pel row is green & blue
0	W	CAP_CTRL	Header Capture Control Writing 1 arms a header capture

Table 2: Control Register Field Definitions

Disabling the core via deassertion of CORE_EN does *not* effect a reset of this or any other register's fields to their hardware reset values. As clarified in heading 3 , reset values denoted by red text are associated exclusively with hardware reset.

Taken together, COL_MODE and ROW_MODE define the expected pattern alignment of *input* pel values presented to the core for processing.

Arming a header capture begins the process of gathering values from the next incoming frame. Software may be apprised of when the next header has been captured by first clearing the IRQF_CAP flag of the IRQ flags register discussed below in heading 3.2 .

3.2 IRQ Flags Register

IRQ_FLAGS_REG

Byte offset 0x10

Bit(s)	[31..1]	0
Access	...	R/C
Name	...	IRQF_CAP
		0

Bit(s)	**Access**	**Name**	**Description**
0	R/C	IRQF_CAP	Flag indicating a header capture has completed. Write '1' to clear.

Table 3: IRQ Flag Register Field Definitions

An asserted IRQ flag has the ability to interrupt the host processor if and only if its corresponding bit IRQM_CAP in the IRQ Mask Register (see heading 3.3) is set as well.

3.3 IRQ Mask Register

IRQ_MASK_REG

Byte offset 0x14

Bit(s)	[31..1]	0
Access	...	R/W
Name	...	IRQM_CAP
		0

Bit(s)	Access	Name	Description
0	R/W	IRQM_CAP	Mask bit for header capture IRQ 0 = Capture IRQ disabled 1 = Capture IRQ enabled

Table 4: IRQ Mask Register Field Definitions

Enabling of any bit within the mask register effectively enables its corresponding IRQ flag register (see heading 3.2) to pass through as an interrupt request to the host processor.

3.4 Resolution Capture Register

RES_CAPTURE_REG

Byte offset 0x20

Bit(s)	[31..16]	[15..0]
Access	R	R
Name	FRM_HEIGHT	FRM_WIDTH
	0xUUUU	0xUUUU

Bit(s)	Access	Name	Description
[31..16]	R	FRM_HEIGHT	Captured header frame height
[15..0]	R	FRM_WIDTH	Captured header frame width

Table 5: Resolution Capture Register Field Definitions

The value of this register is only defined after successful assertion of the IRQF_CAP flag within the Interrupt Flag Register (defined in heading 3.2), as the result of a successful video frame header capture.

3.5 Region-of-Interest (ROI) Capture Register

ROI_CAPTURE_REG

Byte offset 0x24

Bit(s)	[31..16]	[15..0]
Access	R	R
Name	FRM_ROI_Y	FRM_ROI_X
	0xUUUU	0xUUUU

Bit(s)	Access	Name	Description
[31..16]	R	FRM_ROI_Y	Captured header ROI x coordinate
[15..0]	R	FRM_ROI_X	Captured header ROI y coordinate

Table 6: ROI Capture Register Field Definitions

The value of this register is only defined after successful assertion of the IRQF_CAP flag within the Interrupt Flag Register (defined in heading 3.2), as the result of a successful video frame header capture.

3.6 Index Capture Register

INDEX_CAPTURE_REG

Byte offset 0x28

Bit(s)	[31..0]
Access	R
Name	FRM_INDEX
	0xUUUUUUUU

Bit(s)	Access	Name	Description
[31..0]	R	FRM_INDEX	Captured header frame index

Table 7: Index Capture Register Field Definitions

The value of this register is only defined after successful assertion of the IRQF_CAP flag within the Interrupt Flag Register (defined in heading 3.2), as the result of a successful video frame header capture.

3.7 Timestamp Capture Register

TS_CAPTURE_REG

Byte offset 0x2C

Bit(s)	[31..0]
Access	R
Name	FRM_TS
	0xUUUUUUUU

Bit(s)	Access	Name	Description
[31..0]	R	FRM_TS	Captured header frame timestamp

Table 8: Timestamp Capture Register Field Definitions

The value of this register is only defined after successful assertion of the IRQF_CAP flag within the Interrupt Flag Register (defined in heading 3.2), as the result of a successful video frame header capture.

3.8 White Balance [Red] Register

WHITE_BALANCE_RED_REG

Byte offset 0x30

Bit(s)	[31..16]	[15..0]
Access	...	R/W
Name	...	GAIN_RED
		0xUUUU

Bit(s)	**Access**	**Name**	**Description**
[15..0]	R/W	GAIN_RED	White balance gain for red plane

Table 9: White Balance [Red] Register Field Definitions

All white-balance gain values hosted by this register file represent *unsigned* fixed-point quantities, of precision Q1.15. The numerical representation is visually depicted Table by Table:

[15]	**[14:0]**
GAIN_(RED/BLUE/GREEN)	
MANTISSA	FRACTION

Table 10: Numerical Representation of White Balance Gain Values

Valid values for white balance gains are over the domain [0.0, 1.0]; values greater than unity will result in numerical overflow, as no clamping is performed. Any arbitrary floating-point gain value $Gain_{float}$ within this range is mapped (and quantized) to its corresponding fixed-point white balance gain value $Gain_{\text{fixed}}$ as a sixteen (16)-bit integer value by the following formula:

$$Gain_{\text{fixed}} = integer\left(round(Gain_{\text{float}} x 2^{15})\right)$$

3.9 White Balance [Green] Register

WHITE_BALANCE_GREEN_REG

Byte offset 0x34

Bit(s)	[31..16]	[15..0]
Access	...	R/W
Name	...	GAIN_GREEN
		0xUUUU

Bit(s)	**Access**	**Name**	**Description**
[15..0]	R/W	GAIN_GREEN	White balance gain for green plane

Table 11: White Balance [Green] Register Field Definitions

Refer to the table, description, and formula for WHITE_BALANCE_RED_REG, in heading 3.8 , for interpretation of the GAIN_GREEN field.

3.10 White Balance [Blue] Register

WHITE_BALANCE_BLUE_REG

Byte offset 0x38

Bit(s)	[31..16]	[15..0]
Access	...	R/W
Name	...	GAIN_BLUE
		0xUUUU

Bit(s)	**Access**	**Name**	**Description**
[15..0]	R/W	GAIN_BLUE	White balance gain for blue plane

Table 12: White Balance [Blue] Register Field Definitions

Refer to the table, description, and formula for WHITE_BALANCE_RED_REG, in heading 3.8 , for interpretation of the GAIN_BLUE field.

4 Verification Environment

The core is delivered with an automated verification framework implemented atop the Cocotb *(Coroutine Cosimulation Testbench)* environment. A description or primer on Cocotb is beyond the scope of this document; however, the reader is directed and encouraged to refer to reference [2] for further information.

4.1 Test Factory

The simulation makes use of a constrained-randomization layer within Cocotb, which (among other features) automates the creation of a set of randomized tests based upon declarative statements. The set of tests permutes across the vector space of parameters defined as available input test conditions, with the ability to also guide the generation of subsidiary random value generation during the flow of each test.

The test factory approach, combined with a few global control variables, makes it simple to constrain some dimensions for expedience of simulation during development (e.g. fixed resolution, only one Bayer pattern mode), while retaining the ability to specify 'regression mode' for complete test space coverage. The latter is desired when a long simulation time is able to be accommodated for the benefits of greater coverage (e.g. pre-release activities, continuous integration server nightly builds, etc.)

4.2 Golden Model

A "golden model" Python class (`BayerDemosaic`) implements a behavioral model of the core. The model is informed of details related to each test, such as:

* Stimulus video format / resolution

* Input Bayer pattern stimulus configuration

The golden model "understands" the mapping and semantics of the device-under-test (DUT) register file, and performs test initialization of the DUT. This is done in accordance with the randomly-selected test conditions provided by the test factory.

As video stimulus frames are produced, the golden model computes the expected RGB output frame using a numpy-based mathematical transform. Input and output images are stored to the simulation directory as PNG files, and the expected frames of output are placed onto a scoreboard for detailed comparison against DUT output frames.

The stimulus source, scoreboard, and output sink are all subclassed from built-in Cocotb classes. As such, random valid / backpressure capability, etc. are inherited.

4.3 Docker Image

The core is delivered with a Dockerfile capable of independently reproducing the exact set of open-source tools employed to make possible the Cocotb-based verification platform. These include, but are not limited to, the following list:

* Base Ubuntu / Miniconda3 image
* GHDL simulator build
* Cocotb environment build
* Python signal processing libraries (numpy, scipy, PIL, etc.)
* Analysis utilities (GTKWave waveform viewer, EoG image viewer)