LogiCORE IP Color Correction Matrix v3.0

Product Guide

PG001 October 19, 2011

XILINX®

Table of Contents

Appendix B: Migrating

Appendix C: Debugging

Appendix D: Application Software Development

Appendix E: C Model Reference

Appendix F: Additional Resources

LogiCORE IP Color Correction Matrix v3.0

Introduction

The Xilinx LogiCORE™ IP Color Correction Matrix core is a 3 x 3 programmable coefficient matrix multiplier with offset compensation. This core can be used for color correction operations such as adjusting white balance, color cast, brightness, or contrast in an RGB image.

The core efficiently uses the 18x18 bit multipliers, adders and registers provided in XtremeDSP™ slices in Spartan®-6, Virtex®-6, Virtex-7, and Kintex®-7 devices resulting in high performance and optimal resource usage.

Features

- Programmable matrix coefficients
- Selectable processor interface
  - EDK pCore - AXI interface is based on AXI4-Lite specification
  - General Purpose Processor Interface
  - Constant Interface
- Configurable 8-, 10-, and 12-bit input and output
- Independent clipping and clamping control
- Delay match support for up to three sync signals
- Optional CMY input to RGB output color conversion

LogiCORE IP Facts Table	
Core Specifics	
Supported Device Family (1)	Spartan-6, Virtex-6, Virtex-7, Kintex-7
Supported User Interfaces	General Purpose Processor Interface, EDK pCore AXI4-Lite, Constant Interface
Resources	See Table 1-1 through Table 1-4.
Provided with Core	
Documentation	Product Specification
Design Files	Netlists for GPP, Encrypted Source Code for EDK pCore files
Example Design	Not Provided
Test Bench	VHDL
Constraints File	Not Provided
Simulation Models	VHDL or Verilog Structural, C Model (2)
Tested Design Tools	
Design Entry Tools	CORE Generator™ tool, Platform Studio (XPS)
Simulation (3)	Mentor Graphics ModelSim
Synthesis Tools	Xilinx Synthesis Technology (XST) 13.3
Support	
Provided by Xilinx, Inc.	

1. For a complete listing of supported devices, see the release notes for this core.
2. HDL test bench and C Model available on the Product Page on Xilinx.com at http://www.xilinx.com/products/ipcenter/EF-DI-CCM.htm
3. For the supported versions of the tools, see the ISE Design Suite 13: Release Notes Guide.

Chapter 1

Overview

There are many variations that cause difficulties in accurately reproducing color in imaging systems. These include:

- Spectral characteristics of the optics (lens, filters)
- Lighting source variations like daylight, fluorescent, or tungsten
- Characteristics of the color filters of the sensor

The Color Correction Matrix provides a method for correcting the image data for these variations. This fundamental block operates on either CMY or RGB data, and processing is "real-time" as a pre-processing hardware block.

As an example, following one of the three color channels through an imaging system from the original light source to the processed image helps understand the functionality of this core.

The blue color channel is a combination of the blue photons from the scene, multiplied by the relative response of the blue filter, multiplied by the relative response of the silicon to blue photons. However, the filter and silicon responses might be quite different from the response of the human eye, so blue to the sensor is quite different from blue to a human being.

This difference can be corrected and made to more closely match the blue that is acceptable to human vision. The Color Correction Matrix core multiplies the pixel values by some coefficient to strengthen or weaken it, creating an effective gain. At the same time a mixture of green or red can be added to the blue channel. To express this processing mathematically, the new blue (Bc) is related to the old blue (B), red (R), and green (G) according to:

$$Bc = K1 \times R + K2 \times G + K3 \times B$$

where K1, K2, and K3 are the weights for each of the mix of red, green, and blue to the new blue.

Extending this concept, a standard 3 x 3 matrix multiplication can be applied to each of the color channels in parallel simultaneously. This is a matrix operation where the weights define a color-correction matrix. In typical applications, color-correction also contains offset compensation to ensure black [0,0,0] levels are achieved.

$$\begin{bmatrix} R_c \\ G_c \\ B_c \end{bmatrix} = \begin{bmatrix} K_{11} & K_{12} & K_{13} \\ K_{21} & K_{22} & K_{23} \\ K_{31} & K_{32} & K_{33} \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} O_1 \\ O_2 \\ O_3 \end{bmatrix} \quad \text{Equation 1-1}$$

As shown in the matrix operation, the input pixels are transformed to a set of corrected output pixels. This can be a very useful function configured as a static application; however, the programmability of the coefficients and offset values allows this function to adapt to changing lighting conditions based on a separate control loop.

Standards Compliance

AXI4-LITE Interface.

Feature Summary

The Color Correction Matrix core offers a 3x3 matrix multiplication for a variety of color correction applications. The coefficient matrix is fully programmable and includes offset compensation, and clipping and clamping of the output is also definable.

The core offers a processor interface for changing the matrix coefficients during run-time.

Applications

- Pre-processing block for image sensors
- Post-processing core for image data adjustment
- Video surveillance
- Video Conferencing
- Machine Vision

Licensing

The Color Correction Matrix core provides the following three licensing options:

- Simulation Only
- Full System Hardware Evaluation
- Full

After installing the required Xilinx ISE software and IP Service Packs, choose a license option.

Simulation Only

The Simulation Only Evaluation license key is provided with the Xilinx CORE Generator tool. This key lets you assess core functionality with either the example design provided with the Color Correction Matrix core, or alongside your own design and demonstrates the various interfaces to the core in simulation. (Functional simulation is supported by a dynamically generated HDL structural model.)

No action is required to obtain the Simulation Only Evaluation license key; it is provided by default with the Xilinx CORE Generator software.

Full System Hardware Evaluation

The Full System Hardware Evaluation license is available at no cost and lets you fully integrate the core into an FPGA design, place-and-route the design, evaluate timing, and

perform functional simulation of the Color Correction Matrix core using the example design and demonstration test bench provided with the core.

In addition, the license key lets you generate a bitstream from the placed and routed design, which can then be downloaded to a supported device and tested in hardware. The core can be tested in the target device for a limited time before timing out (ceasing to function), at which time it can be reactivated by reconfiguring the device.

To obtain a Full System Hardware Evaluation license, do the following:

1. Navigate to the product page for this core.
2. Click Evaluate.
3. Follow the instructions to install the required Xilinx ISE software and IP Service Packs.

Full

The Full license key is available when you purchase the core and provides full access to all core functionality both in simulation and in hardware, including:

- Functional simulation support
- Full implementation support including place and route and bitstream generation
- Full functionality in the programmed device with no time outs

To obtain a Full license key, you must purchase a license for the core. Click on the "Order" link on the Xilinx.com IP core product page for information on purchasing a license for this core. After doing so, click the "How do I generate a license key to activate this core?" link on the Xilinx.com IP core product page for further instructions.

Installing Your License File

The Simulation Only Evaluation license key is provided with the ISE CORE Generator system and does not require installation of an additional license file. For the Full System Hardware Evaluation license and the Full license, an email will be sent to you containing instructions for installing your license file. Additional details about IP license key installation can be found in the ISE Design Suite Installation, Licensing and Release Notes document.

Performance

Maximum Frequencies

The following are typical clock frequencies for the target devices. The maximum achievable clock frequency can vary. The maximum achievable clock frequency and all resource counts may be affected by other tool options, additional logic in the FPGA device, using a different version of Xilinx tools, or other factors.

- Virtex-7 FPGA: 250 MHz
- Virtex-6 FPGA: 250 MHz
- Kintex-7 FPGA: 250 MHz
- Spartan-6 FPGA: 150 MHz

Latency

The processing latency of the core is seven CLK cycles.

Resource Utilization

For an accurate measure of the usage of primitives, slices, and CLBs for a particular instance, check the **Display Core Viewer after Generation** check box in the CORE Generator interface.

The information presented in Table 1-1 through Table 1-4 is a guide to the resource utilization of the Color Correction Matrix core for all input/output width combinations for Spartan-6, Virtex-6, Virtex-7, and Kintex-7 FPGA families. The Xtreme DSP Slice count is always 9, regardless of parameterization, and this core does not use any block RAMs, dedicated I/O, or CLK resources. The design was tested using ISE® v13.3 tools with default tool options for characterization data.

Table 1-1: **Spartan-6 - xc6slx150,-2 fgg900 (1)**

Input Width	Output Width	LUT6-FF Pairs	LUTs	FFs	DSP48E1	Clock Frequency (MHz)
8	8	190	72	198	9	196
	10	185	77	198	9	196
	12	191	71	198	9	203
10	8	185	87	206	9	173
	10	193	77	206	9	196
	12	193	77	206	9	203
12	8	180	97	214	9	196
	10	196	80	214	9	203
	12	198	78	214	9	196

1. Speedfile: PRODUCTION 1.20c 2011-09-21

Table 1-2: **Virtex-6 xc6vlx75t,-1 ff484(1)**

Input Width	Output Width	LUT6-FF Pairs	LUTs	FFs	DSP48E1	Clock Frequency (MHz)
8	8	179	83	198	9	336
	10	186	76	198	9	336
	12	188	73	198	9	336
10	8	187	81	206	9	342
	10	196	75	206	9	336
	12	185	85	206	9	336
12	8	207	70	214	9	342
	10	200	79	214	9	342
	12	191	85	214	9	342

1. Speedfile: PRODUCTION 1.15 2011-09-21

Table 1-3: **Virtex-7 xc7v585t,-1 ffg1157 (1)**

Input Width	Output Width	LUT6-FF Pairs	LUTs	FFs	DSP48E1	Clock Frequency (MHz)
8	8	180	83	198	9	361
	10	182	81	198	9	352
	12	190	74	198	9	370
10	8	183	88	206	9	370
	10	178	86	206	9	370
	12	187	83	206	9	344
12	8	206	75	214	9	352
	10	208	71	214	9	300
	12	205	75	214	9	326

1. Speedfile: ADVANCED 1.02i 2011-09-21

Table 1-4: **Kintex-7 xc7k70t,-1 fbg484 (1)**

Input Width	Output Width	LUT6-FF Pairs	LUTs	FFs	DSP48E1	Clock Frequency (MHz)
8	8	168	94	198	9	363
	10	182	79	198	9	357
	12	171	87	198	9	363
10	8	188	81	206	9	363
	10	174	97	206	9	357
	12	189	82	206	9	336
12	8	179	96	214	9	371
	10	190	84	214	9	363
	12	203	77	214	9	371

1. Speedfile: ADVANCED 1.02b 2011-09-21

Chapter 2

Core Interfaces and Register Space

Core Symbol and Port Descriptions

The Color Correction Matrix core can be configured with three different interface options, each resulting in a slightly different set of ports. The Color Correction Matrix core uses a set of signals that is common to all of the Xilinx Video IP cores called the Xilinx Streaming Video Interface (XSVI). The XSVI, clk, ce, and sclr signals are common to all interface options and are shown in Figure 2-1 and described by Table 2-1.

Xilinx Streaming Video Interface

The Xilinx Streaming Video Interface (XSVI) is a set of signals common to all of the Xilinx video cores used to stream video data between IP cores. XSVI is also defined as an Embedded Development Kit (EDK) bus type so that the tool can automatically create input and output connections to the core. This definition is embedded in the pCORE interface provided with the IP, and it allows an easy way to cascade connections of Xilinx Video Cores. The Color Correction Matrix IP core uses the following subset of the XSVI signals:

- video_data
- vblank
- hblank
- active_video

Other XSVI signals on the XSVI input bus, such as video_clk, vsync, hsync, field_id, and active_chr do not affect the function of this core.

Note: These signals are neither propagated, nor driven on the XSVI output of this core.

The following is an example EDK Microprocessor Peripheral Definition (.MPD) file definition.

Input side:

```
BUS_INTERFACE BUS = XSVI_IN, BUS_TYPE = TARGET, BUS_STD = XSVI

PORT hblank_in       = hblank,        DIR = I,                          BUS = XSVI_IN
PORT vblank_in       = vblank,        DIR = I,                          BUS = XSVI_IN
PORT active_video_in = active_video,  DIR = I,                          BUS = XSVI_IN
PORT video_data_in   = video_data,    DIR = I, VEC=[3*INPUT_WIDTH-1:0], BUS = XSVI_IN
```

Output side:

```
BUS_INTERFACE BUS = XSVI_OUT, BUS_TYPE = INITIATOR,  BUS_STD = XSVI

PORT hblank_out        = hblank,       DIR = O,                           BUS = XSVI_OUT
PORT vblank_out        = vblank,       DIR = O,                           BUS = XSVI_OUT
PORT active_video_out  = active_video, DIR = O,                           BUS = XSVI_OUT
```

```
PORT video_data_out    = video_data,   DIR = O, VEC=[3*C_OUTPUT_WIDTH-1:0],  BUS = XSVI_OUT
```

The Color Correction Matrix IP core is fully synchronous to the core clock, clk. Consequently, the input XSVI bus is expected to be synchronous to the input clock, clk. Similarly, to avoid clock re-sampling issues, the output XSVI bus for this IP is synchronous to the core clock, clk. The video_clk signals of the input and output XSVI buses are not used.

Constant Interface

The Constant Interface assumes the coefficient matrix and offset values are constants. There is no processor interface and the core is not programmable, but can be reset, enabled, or disabled using the `sclr` and `ce` pins. The ports for the Constant Interface are described in detail in Table 2-1.

As this interface does not provide additional programmability, the Constant Interface has no ports other than the Xilinx Streaming Video Interface, clk, ce, and sclr signals. The Constant Interface Core Symbol is shown in Figure 2-1.

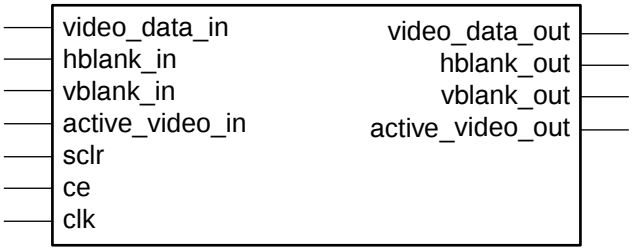

Figure 2-1: **Core Symbol for the Constant Interface**

Table 2-1: **Port Descriptions for the Constant Interface**

Port Name	Port Width	Direction	Description
video_data_in	3**INPUT_WIDTH*	IN	Data input bus
hblank_in	1	IN	Horizontal blanking input
vblank_in	1	IN	Vertical blanking input
active_video_in	1	IN	Active video signal input
video_data_out	3**OUTPUT_WIDTH*	OUT	Data output bus
hblank_out	1	OUT	Horizontal blanking output
vblank_out	1	OUT	Vertical blanking output
active_video_out	1	OUT	Active video signal output
clk	1	IN	Rising-edge clock
ce	1	IN	Clock enable (active high)
sclr	1	IN	Synchronous clear – reset (active high)

- **video_data_in**: This bus contains the three individual color inputs in the following order from MSB to LSB [red : blue : green] or [cyan : yellow : magenta]. Color values are expected in INPUT_WIDTH bits wide unsigned integer representation.

Bits	3*INPUT_WIDTH - 1: 2*INPUT_WIDTH	2*INPUT_WIDTH - 1: INPUT_WIDTH	INPUT_WIDTH - 1:0
Video Data Signals	Red	Blue	Green
	Cyan	Yellow	Magenta

- **hblank_in**: The `hblank_in` signal conveys information about the blank / non-blank regions of video scan lines. This signal is not actively used in the CCM core, but passed through the core with a delay matching the latency of the corrected data.
- **vblank_in**: The `vblank_in` signal conveys information about the blank / non-blank regions of video frames, and is used by the CCM core to detect end of a frame, when user registers can be copied to active registers to avoid visual tearing of the image. This signal is passed through the core with a delay matching the latency of the corrected data.
- **active_video_in**: The `active_video_in` signal is high when valid data is presented at the input. This signal is not actively used in the CCM core, but passed through the core with a delay matching the latency of the corrected data.
- **clk - clock**: Master clock in the design.
- **ce - clock enable**: Pulling `CE` low suspends all operations within the core. Outputs are held, no input signals are sampled, except for reset (`SCLR` takes precedence over `CE`).
- **sclr - synchronous clear**: Pulling `SCLR` high results in resetting all output control signal pins to 0, and resetting the `video_data_out` bus to the clamping values for each color channel. Internal registers within the XtremeDSP slice and D-flip-flops are cleared. However, the core uses SRL16 / SRL32-based delay lines for `hblank`, `vblank`, and `active_video` generation, which are not cleared by `SCLR`. This may result in non-zero outputs after `SCLR` is deasserted, until the contents of SRL16 / SRL32s are flushed. Unwanted results can be avoided if `SCLR` is held active until SRL16 / SRL32s are flushed.
- **video_data_out**: This bus contains RGB output in the same order as `video_data_in`. Color values are represented as OUTPUT_WIDTH bits wide unsigned integers.

Bits	3*OUTPUT_WIDTH - 1: 2*OUTPUT_WIDTH	2*OUTPUT_WIDTH - 1: OUTPUT_WIDTH	OUTPUT_WIDTH - 1:0
Video Data Signals	Red	Blue	Green

- **hblank_out, vblank_out and active_video_out**: The corresponding input signals are delayed so active_video and blanking outputs are in phase with the video data output, maintaining the integrity of the video stream. The blanking and active_video outputs are connected to the corresponding inputs via delay lines matching the propagation delay of the RGB processing pipe. Unwanted blanking inputs should be tied high, and corresponding outputs left unconnected, which will result in the trimming of any unused logic within the core.

  The active_video and blanking signals do not affect the processing behavior of the core. Asserting or deasserting them will not stall processing or the R, G, B streams, and neither will force video outputs to zero.

General Purpose Processor Interface

The General Purpose Processor interface exposes all matrix coefficients and offsets as ports. This option is very useful for developers designing a system with a user-defined bus interface (decoding logic and register banks) to an arbitrary processor.

The Core Symbol for the General Purpose Processor Interface is shown in Figure 2-2. The Xilinx Streaming Video Interface clk, ce, and sclr are described in the previous section. The General Purpose Processor ports are described in Table 2-2.

The coefficient and offset ports have a control mechanism to prevent tearing or committing partially updated port values, and are stored in the port registers within the core. The port values are latched into the working registers at the rising edge of `vblank_in` when bit 1 of `ccm_control` port is set to 1. At the beginning of the next frame, designated by a rising edge in `vblank_in`, the port registers will all update.

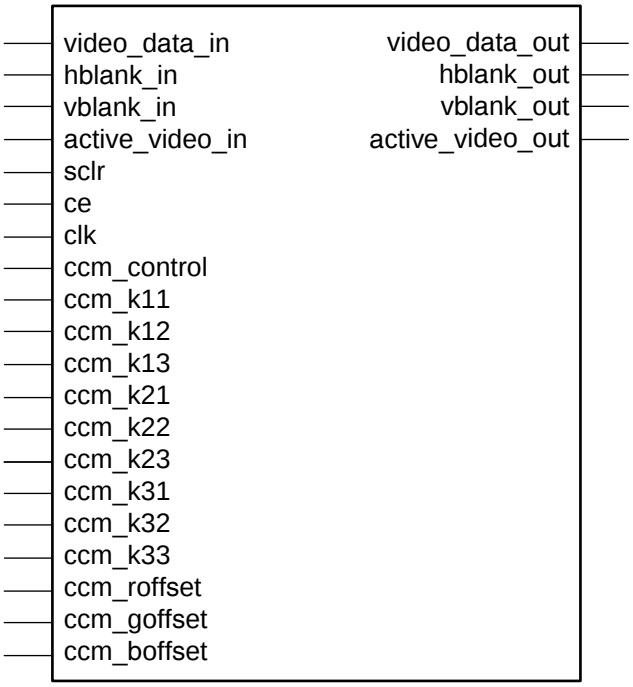

Figure 2-2: **Core Symbol for the General Purpose Processor Interface**

Table 2-2: **Ports for the General Purpose Processor Interface**

Port Name	Port Width	Description
ccm_control	2	Bit 0 Software Enable 0 - Not enabled 1 - Enabled
		Bit 1 Semaphore for AXI4-Lite Register Update 0 - Normal Operating Mode, software safe for updating AXI4-Lite registers 1 - Register Update, stored register values are updated in the core on the next rising edge of `vblank_in`

Table 2-2: **Ports for the General Purpose Processor Interface *(Cont'd)***

Port Name	Port Width	Description
ccm_k11	18	Matrix Coefficient values are presented in 18.15 fixed point format. 18-bit signed integer values are equivalent to real numbers in the [-4 : 4]) range, multiplied by 32768.
ccm_k12	18	
ccm_k13	18	
ccm_k21	18	
ccm_k22	18	
ccm_k23	18	
ccm_k31	18	
ccm_k32	18	
ccm_k33	18	
ccm_roffset	OWIDTH + 1 wide	OWIDTH + 1-bit wide signed integer value in the range [-(2^{OWIDTH}) : (2^{OWIDTH})-1]
ccm_goffset	OWIDTH + 1 wide	
ccm_boffset	OWIDTH + 1 wide	

EDK pCore Interface

The EDK pCore Interface generates AXI4-Lite interface ports listed in Table 2-3, in addition to the clk, ce, sclr, and Xilinx Streaming Video Signals. The AXI4-Lite signals are automatically connected when the generated pCore is inserted into an EDK project. The Core Symbol for the EDK pCore Interface is shown in Figure 2-3. The Xilinx Streaming Video Interface clk, ce, and sclr are described in the previous section (Table 2-1). For more information on the AXI4-Lite bus signals, see UG761, AXI Reference Guide.

Table 2-3: **Ports for the EDK pCore Interface**

Pin Name	Dir	Width	Description
AXI Global System Signals[1]			
S_AXI_ARESETN	I	1	AXI Reset, active low
IP2INTC_Irpt	O	1	Interrupt request output
AXI Write Address Channel Signals[1]			
S_AXI_AWADDR	I	[(C_S_AXI_ADDR_WIDTH-1):0]	AXI4-Lite Write Address Bus. The write address bus gives the address of the write transaction.
S_AXI_AWVALID	I	1	AXI4-Lite Write Address Channel Write Address Valid. This signal indicates that valid write address is available. 1 = Write address is valid. 0 = Write address is not valid.
S_AXI_AWREADY	O	1	AXI4-Lite Write Address Channel Write Address Ready. Indicates core is ready to accept the write address. 1 = Ready to accept address. 0 = Not ready to accept address.

Table 2-3: **Ports for the EDK pCore Interface *(Cont'd)***

Pin Name	Dir	Width	Description
AXI Write Data Channel Signals[1]			
S_AXI_WDATA	I	[(C_S_AXI_DATA_WIDTH-1):0]	AXI4-Lite Write Data Bus.
S_AXI_WSTRB	I	[C_S_AXI_DATA_WIDTH / 8-1:0]	AXI4-Lite Write Strobes. This signal indicates which byte lanes to update in memory.
S_AXI_WVALID	I	1	AXI4-Lite Write Data Channel Write Data Valid. This signal indicates that valid write data and strobes are available. 1 = Write data / strobes are valid. 0 = Write data / strobes are not valid.
S_AXI_WREADY	O	1	AXI4-Lite Write Data Channel Write Data Ready. Indicates core is ready to accept the write data. 1 = Ready to accept data. 0 = Not ready to accept data.
AXI Write Response Channel Signals[1]			
S_AXI_BRESP[2]	O	[1:0]	AXI4-Lite Write Response Channel. Indicates results of the write transfer. 00b = OKAY - Normal access has been successful. 01b = EXOKAY - Not supported. 10b = SLVERR - Error. 11b = DECERR - Not supported.
S_AXI_BVALID	O	1	AXI4-Lite Write Response Channel Response Valid. Indicates response is valid. 1 = Response is valid. 0 = Response is not valid.
S_AXI_BREADY	I	1	AXI4-Lite Write Response Channel Ready. Indicates Master is ready to receive response. 1 = Ready to receive response. 0 = Not ready to receive response.
AXI Read Address Channel Signals[1]			
S_AXI_ARADDR	I	[(C_S_AXI_ADDR_WIDTH-1):0]	AXI4-Lite Read Address Bus. The read address bus gives the address of a read transaction
S_AXI_ARVALID	I	1	AXI4-Lite Read Address Channel Read Address Valid. 1 = Read address is valid. 0 = Read address is not valid.
S_AXI_ARREADY	O	1	AXI4-Lite Read Address Channel Read Address Ready. Indicates core is ready to accept the read address. 1 = Ready to accept address. 0 = Not ready to accept address.

Table 2-3: **Ports for the EDK pCore Interface *(Cont'd)***

Pin Name	Dir	Width	Description
AXI Read Data Channel Signals[1]			
S_AXI_RDATA	O	[(C_S_AXI_DATA_WIDTH-1):0]	AXI4-Lite Read Data Bus.
S_AXI_RRESP[2]	O	[1:0]	AXI4-Lite Read Response Channel Response. Indicates results of the read transfer. 00b = OKAY - Normal access has been successful. 01b = EXOKAY - Not supported. 10b = SLVERR - Error. 11b = DECERR - Not supported.
S_AXI_RVALID	O	1	AXI4-Lite Read Data Channel Read Data Valid. This signal indicates that the required read data is available and the read transfer can complete. 1 = Read data is valid. 0 = Read data is not valid.
S_AXI_RREADY	I	1	AXI4-Lite Read Data Channel Read Data Ready. Indicates master is ready to accept the read data. 1 = Ready to accept data. 0 = Not ready to accept data.

1. The function and timing of these signals are defined in the AMBA AXI Protocol Specification.
2. For signals S_AXI_RRESP[1:0] and S_AXI_BRESP[1:0], the core does not generate the Decode Error ('11') response. Other responses like '00' (OKAY) and '10' (SLVERR) are generated by the core based upon certain conditions.

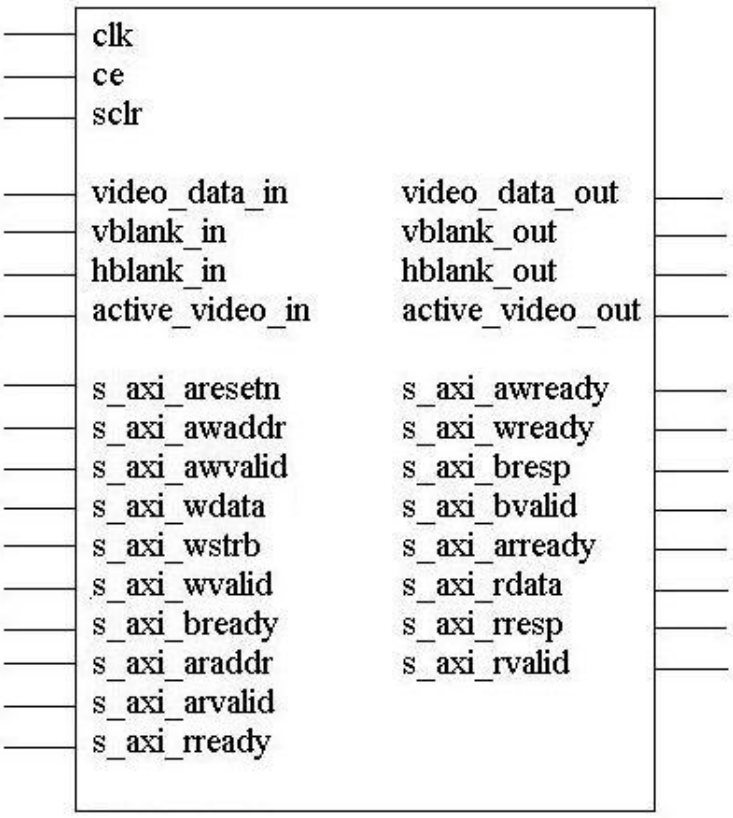

Figure 2-3: **Core Symbol for the EDK pCore Interface**

EDK pCore Interface Register Descriptions

Many imaging applications have an embedded processor that can dynamically control the parameters within the core to correct for variations within the lighting conditions of the scene being captured. The user can select an EDK pCore interface, which creates a pCore that can be added to an EDK project as a hardware peripheral. This pCore provides a memory-mapped interface for the programmable registers of the matrix coefficients and offsets within the core. These registers are described in Table 2-4.

Table 2-4: **EDK pCore Interface Register Descriptions**

Address Offset (hex)	Register Name	Access Type	Default Value (hex)	Description	
0x00	ccm_reg00_control	R/W	0x1	Bit 0	Software Enable 0 – Not enabled 1 – Enabled
				Bit 1	Semaphore for AXI4-Lite Register Update 0 – Normal Operating Mode, software safe for updating AXI4-Lite registers 1 – Register Update, stored register values are updated in the core on the next rising edge of `vblank_in`

Table 2-4: **EDK pCore Interface Register Descriptions *(Cont'd)***

Address Offset (hex)	Register Name	Access Type	Default Value (hex)	Description	
0x04	ccm_reg01_reset	R / W	0x0	Bit 0	Software Manual Reset 0 – Not Reset 1 – Force immediate reset, hold until bit is cleared
				Bit 1	Software Auto-synchronized Reset 0 – Not Reset 1 – Reset occurs automatically on the next rising edge of `vblank_in`
0x08	ccm_reg04_K11	R / W	from GUI	Matrix Coefficient values are presented in 18.15 fixed point format. 18-bit signed integer values are equivalent to real numbers in the [-4 : 4] range, multiplied by 32768.	
0x0C	ccm_reg05_K12	R / W	from GUI		
0x10	ccm_reg06_K13	R / W	from GUI		
0x14	ccm_reg07_K21	R / W	from GUI		
0x18	ccm_reg08_K22	R / W	from GUI		
0x1C	ccm_reg09_K23	R / W	from GUI		
0x20	ccm_reg10_K31	R / W	from GUI		
0x24	ccm_reg11_K32	R / W	from GUI		
0x28	ccm_reg12_K33	R / W	from GUI		
0x2C	ccm_reg13_ROFFSET	R / W	from GUI	OWIDTH + 1-bit wide signed integer value in the range $[-(2^{OWIDTH}) : (2^{OWIDTH})-1]$	
0x30	ccm_reg14_GOFFSET	R / W	from GUI		
0x34	ccm_reg15_BOFFSET	R / W	from GUI		

All of the Write registers are readable, enabling the user to verify writes or read back current values. There is an additional feature that effectively disables the core halting further operations, which blocks the propagation of all video signals. This function is controlled by setting the SW enable register, bit 0 of `ccm_reg00_control`, to 0. The default value of SW enable is 1 (enabled).

All registers other than the control and SW reset registers are double-buffered in hardware to ensure no image tearing happens if the K or offset values are modified in the active area of a frame. This double-buffering provides a more flexible and easier to use core because it decouples the register updates from the blanking period, allowing software a much larger window with which to update the parameter values. The updated values for the K and offset registers are latched into the shadow registers immediately after writing them, while the actual offset and matrix multiplier coefficients used are stored in the working registers.

Any Reads of registers during operation return the values stored in the shadow registers. The rising edge of `vblank_in` triggers the values from the shadow registers to be copied to the working registers, when bit 1 of `ccm_reg00_control` is set to 1. This semaphore bit helps to prevent partially updated shadow registers from being copied over to the working registers.

The EDK pCore option provides a standard AXI4-LiteAXI4-Lite bus interface to the cores.

Importing Generated pCore Into EDK

To generate an EDK pCore, select the EDK pCore option in the CORE Generator Graphical User Interface.

A folder is created in the coregen project directory with the Component Name specified in the CORE Generator Graphical User Interface. This folder contains all of the files needed for an EDK pCore, which contains two folders:

`/drivers`

`/pcores`

Any EDK pCore repository, whether internal or external to an EDK / Xilinx Platform Studio (XPS) project, contains matching `/drivers` and `/pcores` folders containing all of the pCores available in that pCore repository. Copy the contents of these generated folders into your EDK pCore repository location.

For XPS to detect the new pCore, select **Project** → **Rescan user repositories** in XPS.

Once XPS detects the new pCore, it appears in the XPS IP Catalog Tab under the repository in which it has been placed as "Color Correction Matrix 3.00.a". Drag-and-drop, or right-click and select **Add IP** to add a new instance of the core to your project.

To connect the Color Correction Matrix pCore to your AXI4-Lite bus, expand the view of the pCore instance in the System Assembly View's Bus Interfaces tab, and select the associated AXI4-Lite bus in the S_AXI drop-down.

To configure Color Correction Matrix pCore parameters once it has been inserted into your XPS project, right-click the specific instance of the pCore in the System Assembly View window, and select **Configure IP ...** The input and output widths (C_IWIDTH, C_OWIDTH), the clipping (C_MAX) and the clamping values (C_MIN), and AXI4-Lite bus configuration settings are configurable for this pCore instance.

To set the AXI4-Lite base address for the Color Correction Matrix pCore, set the address in either the core's configuration or in the System Assembly View's Addresses' tab. The Color Correction Matrix pCore requires an address range of 256 bytes.

To connect the Color Correction Matrix pCore's video input and output signals, expand the entry for the pCore instance in the System Assembly View's Ports tab. Use the drop-down items to associate each port of the Color Correction Matrix pCore with the desired connection within your EDK system. Note that the clk input is the clock rate of the Color Correction Matrix pCore's video-processing logic, which should be connected to the video clock, not the AXI4-Lite bus clock.

Chapter 3

Customizing and Generating the Core

This chapter includes information on using Xilinx tools to customize and generate the core.

Graphical User Interface (GUI)

The Color Correction Matrix core is easily configured to meet developers' specific needs before instantiation through the CORE Generator™ graphical user interface (GUI). Once developers start to build the Color Correction Matrix core within the CORE Generator system, they are guided through and asked to set various parameters. This section provides a quick reference to the windows and parameters that can be configured at compile time.

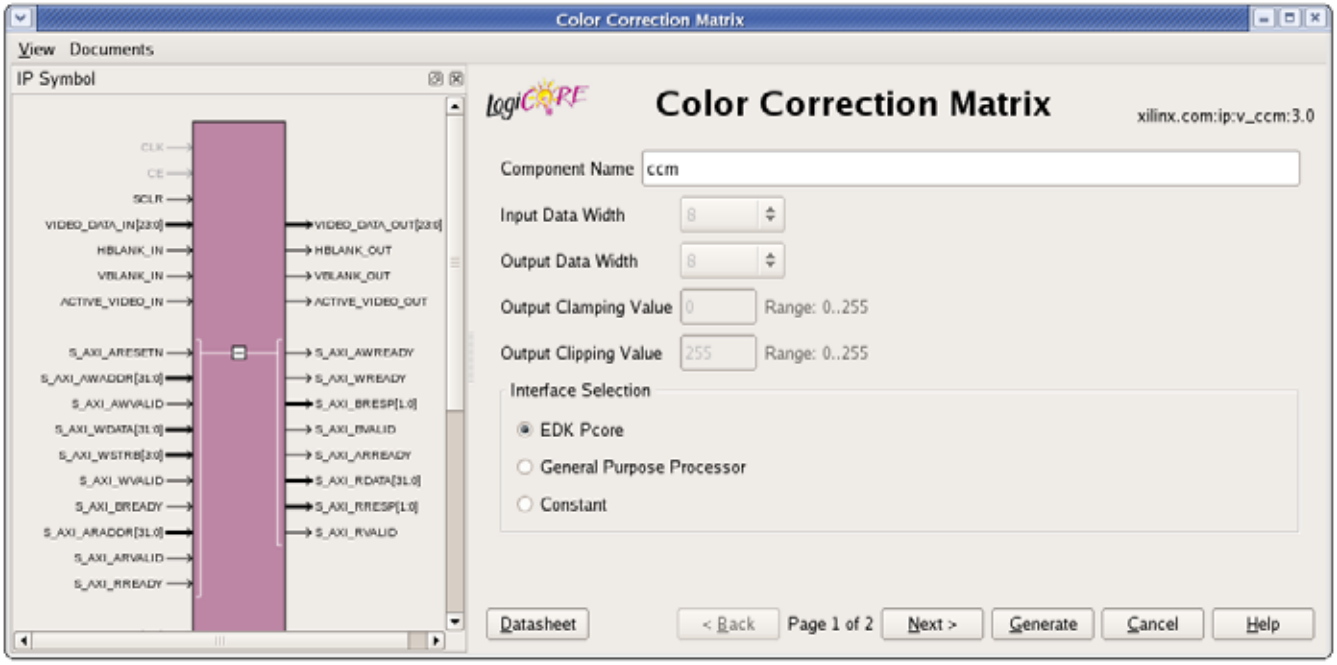

Figure 3-1: **Graphical User Interface – Screen 1**

The first screen (Figure 3-1) shows a representation of the IP symbol on the left side, and the settable parameters on the right, which are described as follows:

- **Component Name**: The component name is used as the base name of output files generated for the module. Names must begin with a letter and must be composed from characters: a to z, 0 to 9 and "_".

- **Input Data Width (IWIDTH)**: Specifies the bit width of the input color channel for each component.
  In cases where the Input Data Width does not equal the Output Data Width, the data is scaled up or down accordingly.
- **Output Data Width (OWIDTH)**: Specifies the bit width of the output color channel for each component.
  In cases where the Input Data Width does not equal the Output Data Width, the data is scaled up or down accordingly.
- **Output Clamping Value**: Specifies the minimum value of the output. Allowable values are from 0 to 2^{OWDITH}-1. The clamping value must be less than the clipping value.
- **Output Clipping Value**: Specifies the maximum value of the output. Allowable values are from 0 to 2^{OWDITH}-1. The clipping value must be larger than the clamping value.
- **Interface Selection**: As described in the previous sections, this option allows for the configuration of three different interfaces for the core.
  - **EDK pCore Interface**: The CORE Generator tool generates a pCore that can be easily imported into an EDK project as a hardware peripheral and coefficients can be programmed via registers. When the EDK pCore interface is selected, the Input- and Output- Data Width, Clipping -, and Clamping Value fields, as well as Page 2 of the GUI are inactive. The parameters of the EDK pCore are controllable via the EDK core configuration GUI and software drivers.
  - **General Purpose Processor Interface**: The CORE Generator tool generates a set of ports to be used to program the core.
  - **Constant Interface**: The matrix coefficients and offsets are constant, and, consequently, no programming is necessary.

The second screen (Figure 3-2) also shows a representation of the IP symbol on the left side, but has a second set of settable parameters on the right, as described in this section.

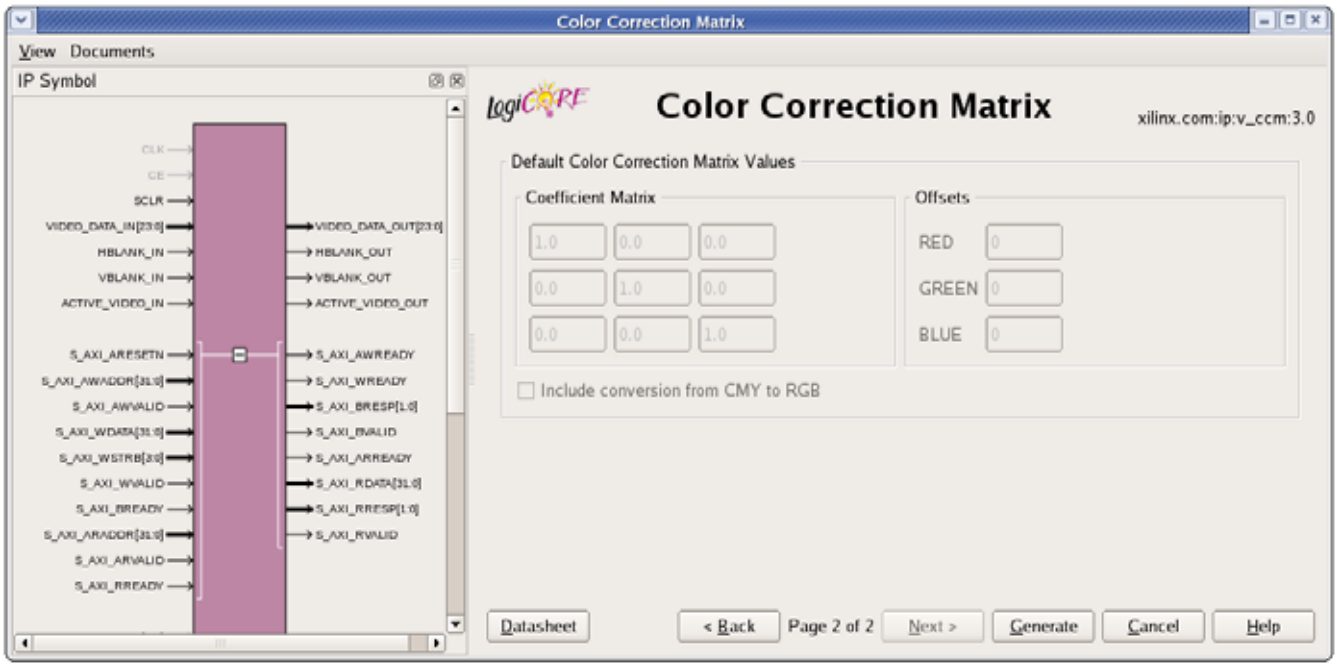

Figure 3-2: **Graphical User Interface – Screen 2**

- **Coefficient Matrix**: Enter the floating-point coefficients ranging from [-4, 4] (K in Equation 1-1) by specifying the 18 bit coefficients with 15 fractional bits of the coefficient matrix. The entered values will be the default used to initialize the core and the values used when the core is reset. Enter the real valued coefficients as floating-point decimal values in the range [-4.0, 4.0] (K in Equation 1-1). When the core is generated, the floating-point decimal value is converted to an 18-bit vector with 15 fractional bits, which are used internally to the core.
- **Offsets**: Enter the offset coefficients (O in Equation 1-1). These signed coefficients have the same bit width as the output. Enter the offset values (O in Equation 1-1). These signed integer values must be in the range $[-2^{OWIDTH}, 2^{OWIDTH}-1]$, and are 1-bit wider than the Output Data Width specified on the first page of the GUI.
- **Include conversion from CMY to RGB**: The Color Correction Matrix core can accept either RGB or CMY inputs. When selecting this conversion, the core will modify the coefficient matrix to also convert the CMY input to RGB output. The equation to calculate the new coefficient matrix is as follows:

$$K' = \begin{bmatrix} K_{11} & K_{12} & K_{13} \\ K_{21} & K_{22} & K_{23} \\ K_{31} & K_{32} & K_{33} \end{bmatrix} \begin{bmatrix} -1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & 1 & -1 \end{bmatrix} \quad \text{Equation 3-1}$$

Parameter Values in the XCO File

Table 1 defines valid entries for the XCO parameters. Xilinx strongly suggests that XCO parameters are not manually edited in the XCO file; instead, use the CORE Generator software GUI to configure the core and perform range and parameter value checking. The XCO parameters are helpful in defining the interface to other Xilinx tools.

Table 3-1: **XCO Parameters**

XCO Parameter	Default	Valid Values
component_name	ccm	ASCII text using characters: a..z, 0..9 and "_" starting with a letter. Note: "v_ccm_v3_0" is not allowed.
interface_selection	EDK_Pcore	EDK_Pcore, General_Purpose Processor, Constant
iwidth	8	8, 10, 12
owidth	8	8, 10, 12
cmy2rgb	false	true, false
k11	1.0	[-4.0, 4.0)
k12	0.0	[-4.0, 4.0)
k13	0.0	[-4.0, 4.0)
k21	0.0	[-4.0, 4.0)
k22	1.0	[-4.0, 4.0)

Table 3-1: **XCO Parameters**

XCO Parameter	Default	Valid Values
k23	0.0	[-4.0, 4.0)
k31	0.0	[-4.0, 4.0)
k32	0.0	[-4.0, 4.0)
k33	1.0	[-4.0, 4.0)
rgb_min	0	0 to $2^{OWIDTH} - 1$
rgb_max	255	0 to $2^{OWIDTH} - 1$
roffset	0	-2^{OWIDTH} to $2^{OWIDTH} - 1$
goffset	0	-2^{OWIDTH} to $2^{OWIDTH} - 1$
boffset	0	-2^{OWIDTH} to $2^{OWIDTH} - 1$

Output Generation

The output files generated from the Xilinx CORE Generator software for the core depend upon whether the interface selection is set to EDK pCore or General Purpose Processor / Constant. The output files are placed in the project directory.

EDK pCore Files

When the interface selection is set to EDK pCore, CORE Generator will output the core as a pCore that can be easily incorporated into an EDK project. The pCore output consists of a hardware pCore and a software driver. The pCore has the following directory structure:

<Component_Name>

- drivers
  - ccm_v3_00_a
    - data
    - build
    - example
    - src
- pcores
  - axi_ccm_v3_00_a
  - data
  - hdl
    - vhdl

File Details

<project directory>

This is the top-level directory. It contains xco and other assorted files.

Name	Description
<component_name>.xco	Log file from CORE Generator software describing which options were used to generate the core. An XCO file can also be used as an input to the CORE Generator software.
<component_name>_flist.txt	A text file listing all of the output files produced when the customized core was generated in the CORE Generator software.

`<project directory>/<component_name>/pcores/axi_ccm_v3_00_a/data`

This directory contains files that EDK uses to define the interface to the pCore.

`< project directory>/<component_name>/pcores/axi_ccm_v3_00_a/hdl/vhdl`

This directory contains the HDL files that implement the pCore.

`< project directory>/<component_name>/drivers/ccm_v3_00_a/data`

This directory contains files that SDK uses to define the operation of the pCore's software driver.

`< project directory>/<component_name>/drivers/ccm_v3_00_a/example`

This directory contains some example code using the pCore's software driver.

`< project directory>/<component_name>/drivers/ccm_v3_00_a/src`

This directory contains the source code of the pCore's software driver.

Name	Description
ccm.c	Provides the API access to all features of the device driver.
ccm.h	Provides the API access to all features of the device driver.

General Purpose Processor or Constant Interface Files

When the interface selection is set to General Purpose Processor, CORE Generator will output the core as a netlist that can be inserted into a processor interface wrapper or instantiated directly in an HDL design. The output is placed in the <project directory>.

File Details

The CORE Generator output consists of some or all the following files.

Table 3-2:

Name	Description
<component_name>_readme.txt	Readme file for the core.
<component_name>.ngc	The netlist for the core.

Table 3-2:

Name	Description
<component_name>.veo	The HDL template for instantiating the core. The structural simulation model for the core. It is used for functionally simulating the core.
<component_name>.vho	
<component_name>.v	
<component_name>.vhd	
<component_name>.xco	Log file from CORE Generator software describing which options were used to generate the core. An XCO file can also be used as an input to the CORE Generator software.
<component_name>_flist.txt	A text file listing all of the output files produced when the customized core was generated in the CORE Generator software.
<component_name>.asy	IP symbol file
<component_name>.gise	ISE subproject files for use when including the core in ISE designs.
<component_name>.xise	

Chapter 4

Designing with the Core

This chapter includes guidelines and additional information to make designing with the core easier.

General Design Guidelines

The Color Correction Matrix core is a 3x3 matrix multiplication with an additional offset.

The output values are:

$Rc = K_{11} \times R + KK_{12} \times G + K_{13} \times B + O_1$

$Gc = K_{21} \times R + K_{22} \times G + K_{33} \times B + O_2$

$Bc = K_{31} \times R + K_{32} \times G + K_{33} \times B + O_3$

In cases where the Input Data Width does not equal the Output Data Width, the data is scaled up or down accordingly. For example, if the Input Data Width =8, and Output Data Width=12, then the core scales the data up by a factor of 4. Meaning, with an identity matrix, an input of 1 will give an output of 4.

Control Signals and Timing

The propagation delay of the Color Correction Matrix core is independent of parameterization and actual signal (R, G, B, `hblank_in`, `vblank_in`, `active_video_in`) values. Deasserting `CE` suspends processing, which may be useful for data-throttling, to temporarily cease processing of a video stream in order to match the delay of other processing components.

The processing latency of the core is seven `CLK` cycles.

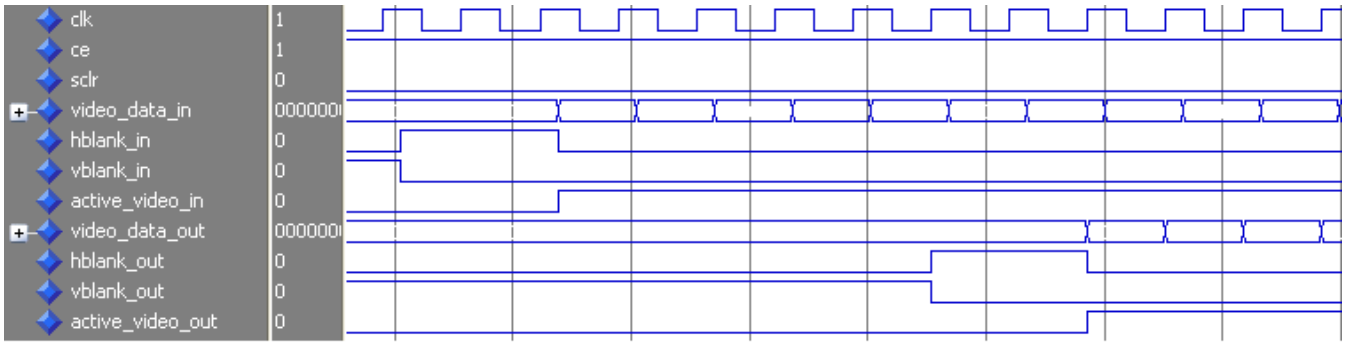

Figure 4-1: **Timing Signals**

XILINX®

Chapter 5

Constraining the Core

Required Constraints

The clk pin should be constrained at the pixel clock rate desired for the video stream.

Device, Package, and Speed Grade Selections

There are no device, package, or speed grade requirements for this core.

Note: The core has not been characterized for use in low power devices.

Clock Frequencies

The clk pin should be run at the required pixel clock frequency for the this core.

Clock Management

There is only one clock for this core.

Clock Placement

There are no specific clock placement requirements for this core

Banking

There are no specific banking rules for this core

Transceiver Placement

There are no transceivers used in this core.

I/O Standard and Placement

There are no specific I/O standard or placement requirements.

Resets

The Color Correction Matrix core has one reset (`sclr`) that is used for the entire core. The reset is `active high`.

Protocol Description

For the pCore version of the Color Correction Matrix core, the register interface is compliant with the AXI4-Lite interface.

XILINX®

Chapter 6

Detailed Example Design

Demonstration Test Bench

Overview

This readme describes how to use the files that come with the demo testbench package for Color Correction Matrix v3.0.

This demo testbench is provided as a simple introductory package that enables core users to observe the core generated by Coregen operating in a waveform simulator. The user is encouraged to observe core-specific aspects in the waveform, make simple modifications to the test conditions, and observe the changes in the waveform.

Software Tools and System Requirements

- Xilinx ISE 13.3 or higher (Includes XST, ISIM, and Coregen).
- ModelSim v6.6d
- ISE Simulator 13.3

Design File Hierarchy

The directory structure underneath this top-level folder is described below:

- Expected
  - Contains the pre-generated expected / golden data used by the testbench to compare actual output data.
- Stimuli
  - Contains the pre-generated input data used by the testbench to stimulate the core (including register programming values).
- Results
  - Actual output data is written to a file in this folder.
- src
  - Contains the .vhd & .xco files of the core.

  The .vhd file is a netlist generated using Coregen.

  You can regenerate a new netlist using the .xco file in Coregen.
- tb_src
  - Contains the top-level testbench design.

  This directory also contains other packages used by the testbench.
- isim_wave.wcfg - Waveform configuration for ISIM
- mti_wave.do - Waveform configuration for ModelSim
- run_isim.bat - Runscript for iSim in Windows OS
- run_isim.sh - Runscript for iSim in Linux OS
- run_mti.bat - Runscript for ModelSim in Windows OS
- run_mti.sh - Runscript for ModelSim in Linux OS

Operating Instructions

- Simulation using ModelSim for Linux:
  From the console, Type "source run_mti.sh".
- Simulation using ModelSim for Windows:
  Double click on "run_mti.bat" file.
- Simulation using iSim for Linux:
  From the console, Type "source run_isim.sh".
- Simulation using iSim for Windows:
  Double click on "run_isim.bat" file.

Support

To obtain technical support for this reference design, go to www.xilinx.com / support to locate answers to known issues in the Xilinx Answers Database or to create a WebCase.

XILINX®

Appendix A

Verification, Compliance, and Interoperability

Simulation

A highly parameterizable test bench was used to test the Color Correction Matrix core. Testing includes the following:

- Register accesses
- Varying matrix coefficients (positive and negative values)
- Varying offset values
- Varying clipping and clamping values
- Testing of various data widths

Hardware Testing

The Color Correction Matrix core has been tested in a variety of hardware platforms at Xilinx to represent a variety of parameterizations.

A test design was developed for the core that incorporated a MicroBlaze processor, AXI4-LITE Interface and various other peripherals.

XILINX®

Appendix B

Migrating

This appendix describes migrating from older versions of the IP to the current IP release.

Functionality Changes

In cases where the Input Data Width does not equal the Output Data Width, the data is scaled accordingly. For example, if the Input Data Width=8, and Output Data Width=12, then the core scales the data up by a factor of 4. Hence, with an identity matrix, an input of 1 yields an output of 4.

Migrating to the EDK pCore AXI4-Lite Interface

The Color Correction Matrix v3.0 interface changed from the PLB processor interface to the EDK pCore AXI4-Lite interface. As a result, all of the PLB-related connections have been replaced with an AXI4-Lite interface. For more information, see the AXI Reference Guide.

XILINX®

Appendix C

Debugging

Evaluation Core Timeout

The Color Correction Matrix v3.0 hardware evaluation core times out after approximately 8 hours of operation. The output is driven to zero. This results in a black screen for RGB color systems and a dark-green screen for YUV color systems.

Application Software Development

Figure D-1 shows a software flow diagram for normal operation and updating registers during the operation of the core. The core can be effectively reset in-system by asserting SW reset (bit 0), which returns all control, coefficient, and offset values to their default values, specified through the Graphical User Interface when the core is instantiated. The core outputs are also forced to 0 until the SW reset bit is deasserted.

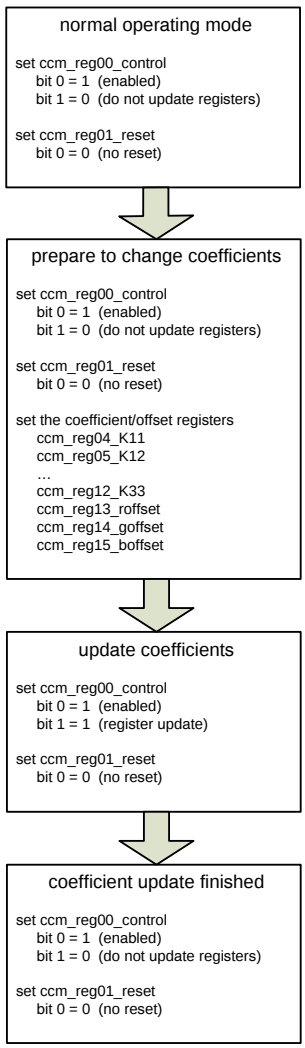

Figure D-1: **Color Correction Matrix Programming Flow Chart**

The software API is provided to allow easy access to the CCM pCore's shared memory registers defined in Table D-1.

To utilize the API functions provided, the following two header files must be included in the user C code:

```
#include "ccm.h"

#include "xparameters.h"
```

The hardware settings of your system, including the base address of your CCM core, are in the `xparameters.h` file. The `ccm.h` file contains the API of functions specifically for controlling the CCM pCore.

The drivers subdirectory of the pCore contains the `example.c` file in the `ccm_v3_00_a/example` subfolder. This file is a sample C program that demonstrates how to use the CCM pCore API. It contains the report_ccm_settings() function, which demonstrates how to use the functions provided by the `ccm.h` driver to read the current status of the core's configuration registers. This file also contains the CCM_Update_Example() function, which provides an example of updating those configuration registers.

Each software register defined in Table D-1 has a constant defined in `ccm.h` that is set to the offset for that register. To write to a register, use the CCM_WriteReg() function using the base address of your CCM pCore instance (from `xparameters.h`), the offset of the desired register, and the data to write. For example:

```
CCM_WriteReg(XPAR_CCM_0_BASEADDR, CCM_REG04_K11, 12345);
```

Reading a value from a register also uses the base address and offset for the register:

```
Xuint32 value = CCM_ReadReg(XPAR_CCM_0_BASEADDR, CCM_REG04_K11);
```

For operations that require reading or writing only a single bit, rather than an entire 32-bit word, `ccm.h` provides pre-defined bit masks as shown in Table D-1.

Table D-1: **Pre-defined Bit Masks**

CCM_CTL_EN_MASK	Bit mask for the software enable bit of the control register
CCM_CTL_RUE_MASK	Bit mask for the register update enable bit of the control register
CCM_RST_RESET	Bit mask for the manual reset bit of the reset register
CCM_RST_AUTORESET	Bit mask for the autoreset bit of the reset register

For additional convenience, pre-defined functions are provided in `ccm.h` for the most-used operations as shown in Table D-2.

Table D-2: **Pre-defined Functions**

CCM_Enable(BaseAddress)	Enables the CCM pCore software enable
CCM_Disable(BaseAddress)	Disables the CCM pCore software enable
CCM_RegUpdateEnable(BaseAddress)	Enables the CCM pCore register update enable
CCM_RegUpdateDisable(BaseAddress)	Disables the CCM pCore register update enable
CCM_Reset(BaseAddress)	Asserts the CCM pCore manual reset
CCM_ClearReset(BaseAddress)	Clears the CCM pCore resets
CCM_AutoSyncReset(BaseAddress)	Asserts the CCM pCore auto reset

Device Drivers

The software API is provided to allow easy access to the pCore's registers defined in Table 7.

To utilize the API functions provided, the following two header files must be included in the user C code:

- `#include "ccm.h"`
- `#include "xparameters.h"`

The hardware settings of your system, including the base address of your core, are defined in the `xparameters.h` file. The `ccm.h` file contains the macro function definitions for controlling the pCore.

For examples on API function calls and integration into a user application, the drivers subdirectory of the pCore contains a file, example.c, in the ccm_v3_00_a/example subfolder. This file is a sample C program that demonstrates how to use the pCore API.

EDK pCore API Functions

This section describes the functions included in the C driver (ccm.c and ccm.h) generated for the EDK pCore API.

CCM_Enable(uint32 BaseAddress);

This macro enables a Color Correction Matrix instance.

BaseAddress is the Xilinx EDK base address of the Color Correction Matrix core (from xparameters.h).

CCM_Disable(uint32 BaseAddress);

This macro disables a Color Correction Matrix instance.

BaseAddress is the Xilinx EDK base address of the Color Correction Matrix core (from xparameters.h).

CCM_Reset(uint32 BaseAddress);

This macro resets a Color Correction Matrix instance. This reset effects the core immediately, and may cause image tearing. Reset affects the coefficient and frame size registers, forces video_data_out to 0, and forces timing signal outputs to their reset state until CCM_ClearReset() is called.

BaseAddress is the Xilinx EDK base address of the Color Correction Matrix core (from xparameters.h)

CCM_ClearReset(uint32 BaseAddress);

This macro clears the reset flag of the core, which allows it to re-sync with the input video stream and return to normal operation.

BaseAddress is the Xilinx EDK base address of the Color Correction Matrix core (from xparameters.h).

CCM_RegUpdateEnable(uint32 BaseAddress);

Calling RegUpdateEnable causes the Color Correction Matrix to start using the updated register values on the next rising edge of vblank_in. The user must manually disable the register update after a sufficient amount of time to prevent continuous updates.

This function only works when the Color Correction Matrix core is enabled.

BaseAddress is the Xilinx EDK base address of the Color Correction Matrix core (from xparameters.h)

CCM_RegUpdateDisable(uint32 BaseAddress);

Disabling the Register Update prevents the Color Correction Matrix gain registers from updating. It is recommended that the Register Update be disabled while writing to the registers in the core, until the write operation is complete. While disabled, writes to the registers are stored, but do not affect the core's behavior.

This function only works when the Color Correction Matrix core is enabled.

BaseAddress is the Xilinx EDK base address of the Color Correction Matrix core (from xparameters.h)

XILINX®

Appendix E

C Model Reference

The Xilinx® LogiCORE™ IP Color Correction Matrix core has a bit accurate C model designed for system modeling.

Features

- Bit accurate with the CCM v3.0 core
- Statically linked library (.lib, .o, .obj – Windows)
- Dynamically linked library (.so – Linux)
- Available for 32 and 64-bit Windows and Linux platforms
- Supports all features of the CCM core that affect numerical results
- Designed for rapid integration into a larger system model
- Example C code is provided to show how to use the function
- Example application C code wrapper file supports 8-bit BMP only

Overview

The Xilinx LogiCORE IP CCM v3.0 has a bit accurate C model for 32- and 64-bit Linux platforms. The model has an interface consisting of a set of C functions, which reside in a statically link library (shared library). Full details of the interface are given in Using the C Model. An example piece of C code is provided to show how to call the model.

The model is bit accurate, as it produces exactly the same output data as the core on a frame-by-frame basis. However, the model is not cycle accurate, as it does not model the core's latency or its interface signals.

The latest version of the model is available for download on the Xilinx™ LogiCORE IP CCM product page at:

http://www.xilinx.com/products/intellectual-property/EF-DI-CCM.htm

Technical Support

For technical support, go to www.xilinx.com/support. Questions are routed to a team with expertise using the CCM v3.0 core. Xilinx provides technical support for use of this product as described in this product guide.

Xilinx cannot guarantee functionality or support of this product for designs that do not follow these guidelines.

Feedback

Xilinx welcomes comments and suggestions about the CCM v3.0 core and the accompanying documentation.

CCM v3.0 Bit Accurate C Model and IP Core

For comments or suggestions about the CCM v3.0 core and bit accurate C model, submit a WebCase from: http://www.xilinx.com/support/clearexpress/websupport.htm

Be sure to include this information:

- Product name
- Core version number
- Explanation of your comments

Document

For comments or suggestions about the documentation for the CCM v3.0 core and bit accurate C model, submit a WebCase from:

http://www.xilinx.com/support/clearexpress/websupport.htm

Be sure to include this information:

- Document title
- Document number
- Page number(s) to which your comments refer
- Explanation of your comments

Unpacking and Model Contents

Unzip the `v_ccm_v3_0_bitacc_model.zip` file, containing the bit accurate models for the Color Correction Matrix IP Core. This creates the directory structure and files in Table E-1.

Table E-1: **Directory Structure and Files of the CCM v3.0 Bit Accurate C Model**

File Name	Contents
README.txt	Release notes
pg001_v_ccm.pdf	LogiCORE IP Color Correction Matrix Product Guide
v_ccm_v3_0_bitacc_cmodel.h	Model header file
rgb_utils.h	Header file declaring the RGB image/video container type and support functions
bmp_utils.h	Header file declaring the bitmap (.bmp) image file I/O functions
video_utils.h	Header file declaring the generalized image/video container type, I/O and support functions
run_bitacc_cmodel.c	Example code calling the C model
kodim19_128x192.bmp	128x192 sample test image of the lighthouse image from the True Color Kodak test images

Table E-1: **Directory Structure and Files of the CCM v3.0 Bit Accurate C Model *(Cont'd)***

/lin64	Precompiled bit accurate ANSI C reference model for simulation on 64-bit Linux platforms
libIp_v_ccm_v3_0_bitacc_cmodel.so	Model shared object library
libstlport.so.5.1	STL library, referenced by libIp_v_ccm_v3_0_bitacc_cmodel.so
/nt32	Precompiled bit accurate ANSI C reference model for simulation on 32-bit Windows platforms.
libIp_v_ccm_v3_0_bitacc_cmodel.lib	Precompiled library file for win32 compilation
/lin32	Precompiled bit accurate ANSI C reference model for simulation on 32-bit Linux platforms
libIp_v_ccm_v3_0_bitacc_cmodel.so	Model shared object library
libstlport.so.5.1	STL library, referenced by libIp_v_ccm_v3_0_bitacc_cmodel.so
/nt64	Precompiled bit accurate ANSI C reference model for simulation on 64-bit Windows platforms.
libIp_v_ccm_v3_0_bitacc_cmodel.lib	Precompiled library file for nt64 compilation

Installation

For Linux, make sure these files are in a directory that is in your $LD_LIBRARY_PATH environment variable:

- `libIp_v_ccm_v3_0_bitacc_cmodel.so`
- `libstlport.so.5.1`

Software Requirements

The CCM v3.0 C models were compiled and tested with the software listed in Table E-2.

Table E-2: **Compilation Tools for the Bit Accurate C Models**

Platform	C Compiler
32- and 64-bit Linux	GCC 4.1.1
32- and 64-bit Windows	Microsoft Visual Studio 2005

Using the C Model

The bit accurate C model is accessed through a set of functions and data structures that are declared in the `v_ccm_v3_0_bitacc_cmodel.h` file.

Before using the model, the structures holding the inputs, generics and output of the CCM instance must be defined:

```
struct  xilinx_ip_v_ccm_v3_0_generics ccm_generics;
struct  xilinx_ip_v_ccm_v3_0_inputs   ccm_inputs;
struct  xilinx_ip_v_ccm_v3_0_outputs  ccm_outputs;
```

The declaration of these structures is in the `v_ccm_v3_0_bitacc_cmodel.h` file.

Table E-3 lists the generic parameters taken by the CCM v3.0 IP core bit accurate model, as well as the default values. For an actual instance of the core, these parameters can only be set in generation time through the CORE Generator™ GUI.

Table E-3: **Model Generic Parameters and Default Values**

Generic variable	Type	Default Value	Range	Description
IWIDTH	int	8	8,10,12	Input data width
OWIDTH	int	8	8,10,12	Output width

Calling `xilinx_ip_v_ccm_v3_0_get_default_generics(&ccm_generics)` initializes the generics structure with the CCM GUI defaults, listed in Table E-3.

Coefficients, offsets, clipping and clamping values can also be set dynamically through the pCore and General Purpose Processor interfaces. Consequently, these values are passed as inputs to the core, along with the actual test image, or video sequence (Table E-4).

Table E-4: **Core Generic Parameters and Default Values**

Input Variable	Type	Default Value	Range(1)	Description
video_in	video_struct	null	N / A	Container to hold input image or video data.[2]
coeffs	double[3][3]	identity[1]	[-4 to 4]	3x3 matrix of floating point numbers
offsets	double[3]	zeros[1]	-2^{OWIDTH} *to* $2^{OWIDTH}-1$	Offsets applied to the output color channels
CLAMP	int	0	0 *to* $2^{OWIDTH}-1$	Clamping value for outputs
CLIP	int	$2^{OWIDTH}-1$	0 *to* $2^{OWIDTH}-1$	Clipping value for outputs

1. OWIDTH is the output data width of each color component

[1] For a detailed description of inputs and other generic parameters, see Core Interfaces and Register Space.
[2] For the description of the input structure, see Initializing the CCM Input Video Structure.

The structure `ccm_inputs` defines the values of run time parameters and the actual input image. Calling `xilinx_ip_v_ccm_v3_0_get_default_inputs(&ccm_generics, &ccm_inputs)` initializes the input structure with the CCM GUI default values (see Table E-4).

Note: The `video_in` variable is not initialized because the initialization depends on the actual test image to be simulated. Initializing the CCM Input Video Structure describes the initialization of the `video_in` structure.

After the inputs are defined, the model can be simulated by calling this function:

```
int xilinx_ip_v_ccm_v3_0_bitacc_simulate(
struct xilinx_ip_v_ccm_v3_0_generics* generics,
struct xilinx_ip_v_ccm_v3_0_inputs*   inputs,
struct xilinx_ip_v_ccm_v3_0_outputs*  outputs).
```

Results are included in the outputs structure, which contains only one member, type `video_struct`. After the outputs are evaluated and saved, dynamically allocated memory for input and output video structures must be released by calling this function:

```
void xilinx_ip_v_ccm_v3_0_destroy(
```

```
struct xilinx_ip_v_ccm_v3_0_inputs *input,
struct xilinx_ip_v_ccm_v3_0_outputs *output).
```

Successful execution of all provided functions, except for the destroy function, return value 0. A non-zero error code indicates that problems occurred during function calls.

CCM Input and Output Video Structure

Input images or video streams can be provided to the CCM v3.0 reference model using the `video_struct` structure, defined in `video_utils.h`:

```
struct video_struct{
  int       frames, rows, cols, bits_per_component, mode;
  uint16*** data[5]; };
```

Table E-5: **Member Variables of the Video Structure**

Member Variable	Designation
frames	Number of video / image frames in the data structure.
rows	Number of rows per frame. Pertaining to the image plane with the most rows and columns, such as the luminance channel for YUV data. Frame dimensions are assumed constant through all frames of the video stream. However different planes, such as y, u and v can have different dimensions.
cols	Number of columns per frame. Pertaining to the image plane with the most rows and columns, such as the luminance channel for YUV data. Frame dimensions are assumed constant through all frames of the video stream. However different planes, such as y, u and v can have different dimensions.
bits_per_component	Number of bits per color channel / component.All image planes are assumed to have the same color / component representation. Maximum number of bits per component is 16.
mode	Contains information about the designation of data planes. Named constants to be assigned to mode are listed in Table E-6.
data	Set of five pointers to three dimensional arrays containing data for image planes. Data is in 16-bit unsigned integer format accessed as data[plane][frame][row][col].

Table E-6: **Named Video Modes with Corresponding Planes and Representations**

Mode(1)	Planes	Video Representation
FORMAT_MONO	1	Monochrome – Luminance only
FORMAT_RGB	3	RGB image / video data
FORMAT_C444	3	444 YUV, or YCrCb image / video data
FORMAT_C422	3	422 format YUV video, (u, v chrominance channels horizontally sub-sampled)

Table E-6: **Named Video Modes with Corresponding Planes and Representations**

FORMAT_C420	3	420 format YUV video, (u, v sub-sampled both horizontally and vertically)
FORMAT_MONO_M	3	Monochrome (Luminance) video with Motion
FORMAT_RGBA	4	RGB image / video data with alpha (transparency) channel
FORMAT_C420_M	5	420 YUV video with Motion
FORMAT_C422_M	5	422 YUV video with Motion
FORMAT_C444_M	5	444 YUV video with Motion
FORMAT_RGBM	5	RGB video with Motion

1. The Color Correction Matrix core supports Modes FORMAT_RGB and FORMAT_C444.

Initializing the CCM Input Video Structure

The easiest way to assign stimuli values to the input video structure is to initialize it with an image or video. The `bmp_util.h` and `video_util.h` header files packaged with the bit accurate C models contain functions to facilitate file I/O.

Bitmap Image Files

The header `bmp_utils.h` declares functions that help access files in Windows Bitmap format (http://en.wikipedia.org/wiki/BMP_file_format). However, this format limits color depth to a maximum of 8-bits per pixel, and operates on images with three planes (R,G,B). Consequently, the following functions operate on arguments type `rgb8_video_struct`, which is defined in `rgb_utils.h`. Also, both functions support only true-color, non-indexed formats with 24-bits per pixel.

```
int write_bmp(FILE *outfile, struct rgb8_video_struct *rgb8_video);
int read_bmp(FILE *infile, struct rgb8_video_struct *rgb8_video);
```

Exchanging data between `rgb8_video_struct` and general `video_struct` type frames / videos is facilitated by these functions:

```
int copy_rgb8_to_video(struct rgb8_video_struct* rgb8_in,
                       struct video_struct* video_out );
int copy_video_to_rgb8(struct video_struct* video_in,
                       struct rgb8_video_struct* rgb8_out );
```

Note: All image/video manipulation utility functions expect both input and output structures initialized; for example, pointing to a structure that has been allocated in memory, either as static or dynamic variables. Moreover, the input structure must have the dynamically allocated container (data or r, g, b) structures already allocated and initialized with the input frame(s). If the output container structure is pre-allocated at the time of the function call, the utility functions verify and issue an error if the output container size does not match the size of the expected output. If the output container structure is not pre-allocated, the utility functions create the appropriate container to hold results.

Binary Image/Video Files

The `video_utils.h` header file declares functions that help load and save generalized video files in raw, uncompressed format.

```
int read_video( FILE* infile,  struct video_struct* in_video);
int write_video(FILE* outfile, struct video_struct* out_video);
```

These functions serialize the video_struct structure. The corresponding file contains a small, plain text header defining, "Mode", "Frames", "Rows", "Columns", and "Bits per Pixel". The plain text header is followed by binary data, 16-bits per component in scan line continuous format. Subsequent frames contain as many component planes as defined by the video mode value selected. Also, the size (rows, columns) of component planes can differ within each frame as defined by the actual video mode selected.

Working with Video_struct Containers

The video_utils.h header file defines functions to simplify access to video data in video_struct.

```
int video_planes_per_mode(int mode);
int video_rows_per_plane(struct video_struct* video, int plane);
int video_cols_per_plane(struct video_struct* video, int plane);
```

The video_planes_per_mode function returns the number of component planes defined by the mode variable, as described in Table E-6. The video_rows_per_plane and video_cols_per_plane functions return the number of rows and columns in a given plane of the selected video structure. The following example demonstrates using these functions in conjunction to process all pixels within a video stream stored in the in_video variable:

```
for (int frame = 0; frame < in_video->frames; frame++) {
  for (int plane = 0; plane < video_planes_per_mode(in_video->mode); plane++) {
    for (int row = 0; row < rows_per_plane(in_video,plane); row++) {
      for (int col = 0; col < cols_per_plane(in_video,plane); col++) {
   // User defined pixel operations on
// in_video->data[plane][frame][row][col]
      }
    }
  }
}
```

C Model Example Code

An example C file, run_bitacc_cmodel.c, is provided to demonstrate the steps required to run the model. After following the compilation instructions, run the example executable. The executable takes the path / name of the input file and the path / name of the output file as parameters. If invoked with insufficient parameters, this help message is issued:

```
Usage: run_bitacc_cmodel in_file out_file
in_file     : path/name of the input  BMP file
out_file    : path/name of the output BMP file
```

During successful execution, two files with a .bin extension are created. The first file corresponds to the input BMP image, with the same path and name as the input file, and a .bin extension. The other file similarly corresponds to the output file. These files contain the inputs and outputs of the CCM algorithm in full precision, as the BMP format does not support color resolutions beyond 8-bits per component. The structure of .bin files are described in Binary Image / Video Files.

To ease modifying and debugging the provided top-level demonstrator using the built-in debugging environment of Visual Studio, the top-level command line parameters can be specified through the Project Property Pages using these steps:

1. In the Solution Explorer pane, right-click the project name and select Properties in the context menu.
2. Select Debugging on the left pane of the Property Pages dialog box.
3. Enter the paths and file names of the input and output images in the Command Arguments field.

Compiling CCM C Model with Example Wrapper

Linux (32- and 64-bit)

To compile the example code, perform these steps:

1. Set your `$LD_LIBRARY_PATH` environment variable to include the root directory where you unzipped the model zip file using a command such as:

   ```
   setenv LD_LIBRARY_PATH <unzipped_c_model_dir>:${LD_LIBRARY_PATH}
   ```

2. Copy these files from the /lin64 directory to the root directory:

   `libstlport.so.5.1`

   `libIp_v_ccm_v3_0_bitacc_cmodel.so`

3. In the root directory, compile using the GNU C Compiler with this command:

   ```
   gcc -x c++ run_bitacc_cmodel.c -o run_bitacc_cmodel -L.
   -lIp_v_ccm_v3_0_bitacc_cmodel -Wl,-rpath,.
   ```

Windows (32- and 64-bit)

The precompiled library `v_ccm_v3_0_bitacc_cmodel.lib`, and top-level demonstration code `run_bitacc_cmodel.c` should be compiled with an ANSI C compliant compiler under Windows. An example procedure is provided here using Microsoft Visual Studio.

1. In Visual Studio, create a new, empty Win32 Console Application project.
2. As existing items, add:
   a. `libIp_v_ccm_v3_0_bitacc_cmodel.lib` to the Resource Files folder of the project
   b. `run_bitacc_cmodel.c` to the Source Files folder of the project
   c. `v_ccm_v3_0_bitacc_cmodel.h` to the Header Files folder of the project
3. After the project is created and populated, it must be compiled and linked (built) to create a win32 executable. To perform the build step, select "Build Solution" from the Build menu. An executable matching the project name has been created either in the Debug or Release subdirectories under the project location based on whether "Debug" or "Release" has been selected in the "Configuration Manager" under the Build menu.

XILINX®

Appendix F

Additional Resources

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see the Xilinx Support website at:

http://www.xilinx.com/support.

For a glossary of technical terms used in Xilinx documentation, see:

http://www.xilinx.com/support/documentation/sw_manuals/glossary.pdf.

References

These documents provide supplemental material useful with this user guide:

- AXI Reference Guide.
- AMBA AXI4 Interface Protocol.

Technical Support

Xilinx provides technical support at www.xilinx.com/support for this LogiCORE™ IP product when used as described in the product documentation. Xilinx cannot guarantee timing, functionality, or support of product if implemented in devices that are not defined in the documentation, if customized beyond that allowed in the product documentation, or if changes are made to any section of the design labeled DO NOT MODIFY.

See the IP Release Notes Guide (XTP025) for more information on this core. For each core, there is a master Answer Record that contains the Release Notes and Known Issues list for the core being used. The following information is listed for each version of the core:

- New Features
- Resolved Issues
- Known Issues

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
10/19/2011	1.0	Initial Xilinx release.

Notice of Disclaimer